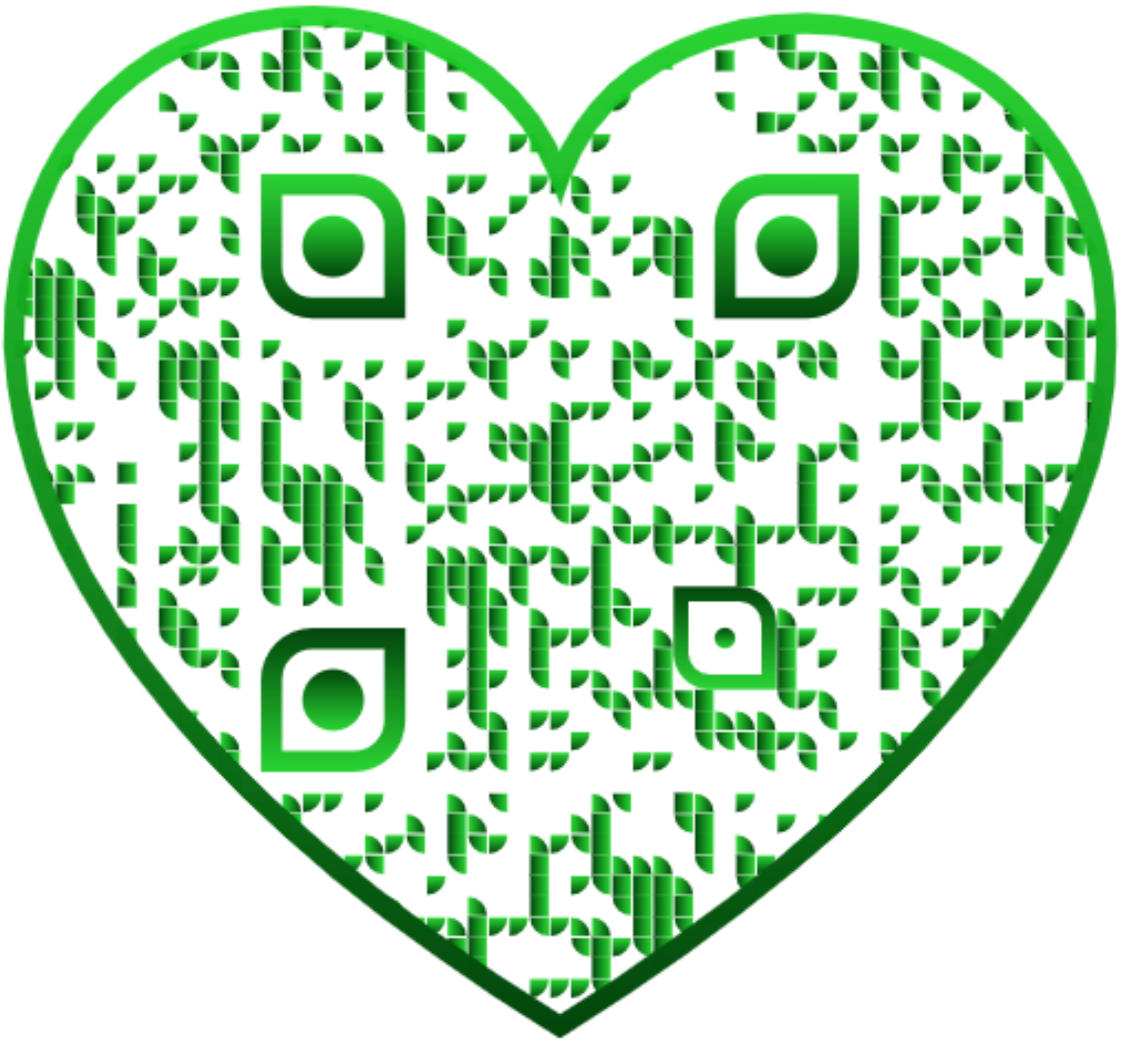
Thomas H. Cormen

# Algorithmes

## Notions de base

Licence, IUT
Écoles d'ingénieurs

DUNOD



L'édition originale de ce livre a été publiée aux États-Unis
par The MIT Press, Cambridge, Massachussets,
sous le titre *Algorithms Unlocked*.
Copyright © 2013 Massachussets Institute of Technology

Traduit de l'américain par Hervé Soulard

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

Illustration de couverture : © Droits réservés

© Dunod, Paris, 2013 pour la traduction française
ISBN 978-2-10-070290-9

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Copyright © 2013 Dunod.

# Table des matières


Avant-propos VI

**Chapitre 1.** Introduction aux algorithmes 1

- Exactitude 2
- Utilisation des ressources 4
- Algorithmes pour les non-informaticiens 6
- Algorithmes pour les informaticiens 6
- Lectures complémentaires 8

**Chapitre 2.** Décrire et évaluer des algorithmes 11

- Décrire des algorithmes informatiques 11
- Caractériser les temps d'exécution 18
- Invariants de boucle 22
- Récursion 23
- Lectures complémentaires 25

**Chapitre 3.** Algorithmes de tri et de recherche 27

- Recherche dichotomique 30
- Tri par sélection 34
- Tri par insertion 37
- Tri par fusion 41
- Tri rapide 50
- Récapitulatif 58
- Lectures complémentaires 61



Copyright © 2013 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

**Chapitre 4.** Minorant pour le tri à battre 63

Règles pour le tri 63

Minorant pour le tri par comparaison 64

Dépasser le minorant avec le tri par dénombrement 65

Tri par base 71

Lectures complémentaires 73

**Chapitre 5.** Graphes orientés acycliques 75

Graphes orientés acyliques 78

Tri topologique 79

Représenter un graphe orienté 82

Temps d'exécution du tri topologique 84

Chemin critique dans un diagramme PERT 85

Plus court chemin dans un graphe orienté acyclique 89

Lectures complémentaires 93

**Chapitre 6.** Plus courts chemins 95

Algorithme de Dijkstra 97

Algorithme de Bellman-Ford 106

Algorithme de Floyd-Warshall 111

Lectures complémentaires 118

**Chapitre 7.** Algorithmes sur les chaînes de caractères 121

Plus longue sous-séquence commune 122

Transformer une chaîne en une autre 127

Recherche de chaîne de caractères 135

Lectures complémentaires 142

**Chapitre 8.** Bases de la cryptographie 143

Chiffrement par simple substitution 144

Cryptographie à clé symétrique 146

Cryptographie à clé publique 149

Copyright © 2013 Dunod.

Cryptosystème RSA 151
Cryptosystèmes hybrides 160
Générer des nombres aléatoires 160
Lectures complémentaires 161

**Chapitre 9. Compression de données 163**

Codages de Huffman 165
Télécopieurs 171
Compression LZW 172
Lectures complémentaires 183

**Chapitre 10. Complexité 185**

Camions marron 185
Classes de complexité N et NP, et NP-complétude 189
Problèmes de décision et réductions 190
Un problème mère 194
Échantillon de problèmes NP-complets 195
Stratégies générales 211
Perspectives 214
Problèmes indécidables 216
Conclusion 218
Lectures complémentaires 219

**Bibliographie 220**

**Index 221**

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

# AVANT-PROPOS

Comment les ordinateurs font-ils pour résoudre des problèmes ? Comment votre petit GPS parvient-il à déterminer, parmi la multitude d'itinéraires possibles, le chemin le plus rapide vers votre destination, et cela en quelques secondes ? Lors d'un achat sur Internet, comment votre numéro de carte bancaire est-il protégé contre quiconque l'intercepterait ? La réponse à ces questions, comme à de nombreuses autres, tient dans les *algorithmes*. Ce livre va vous dévoiler les mystères des algorithmes.

Je compte parmi les auteurs de l'ouvrage *Algorithmique*. Si ce livre est vraiment très bien (peut-être suis-je un tantinet partisan), il peut être trop technique sur certains points.

Celui-ci est différent. Ce n'est pas un manuel pour étudiant « classique ». Il n'aborde pas le domaine des algorithmes informatiques de façon générale ni détaillée. Il n'enseigne pas de façon normative des techniques de conception des algorithmes. Il ne propose au lecteur aucun problème ou exercice.

Dans ce cas, qu'allez-vous y trouver ?

Cet ouvrage sera un point de départ si :

- vous voulez comprendre comment les ordinateurs résolvent des problèmes ;
- vous souhaitez savoir comment évaluer la qualité de ces solutions ;
- vous aimeriez mettre en rapport les approches employées dans la résolution des problèmes informatiques et le monde extérieur ;
- vous êtes capable de lire des expressions mathématiques finalement pas si compliquées ;
- vous n'avez pas nécessairement déjà écrit un programme informatique (mais l'avoir fait ne sera pas rédhibitoire).

Certains ouvrages qui traitent des algorithmes prennent une approche conceptuelle, avec peu de détails techniques. D'autres affichent une précision technique stupéfiante. D'autres encore trouvent leur place entre ces deux extrêmes. C'est le cas de celui-ci. Oui vous y trouverez des éléments mathématiques. Oui certains points seront abordés avec une grande

Copyright © 2013 Dunod.

précision. Cependant, je vais éviter d'aller trop loin dans les détails, excepté peut-être vers la fin de l'ouvrage, là où j'ai un peu perdu le contrôle.

Je vois ce livre comme l'entrée d'un repas. Supposons que vous alliez au restaurant et que vous commandiez l'entrée. Avant de commander la suite, vous attendez qu'elle arrive et que vous la mangiez. Si vous ne l'aimez pas, vous décidez que le repas est terminé. Peut-être vous a-t-elle plu, mais elle a suffi à vous rassasier. Ou bien vous l'avez appréciée, mais vous avez encore faim et décidez de passer à la suite. En lisant cet ouvrage, je souhaite que vous arriviez dans l'un des deux derniers cas : vous êtes satisfait mais sans avoir envie d'aller plus loin dans le monde des algorithmes, ou vous avez tellement apprécié le contenu que vous souhaitez en apprendre plus. Chaque chapitre se termine par une section intitulée « Lectures complémentaires » qui propose d'autres ouvrages et articles relatifs au sujet traité.

## Objectif du livre

Je ne suis pas en mesure de vous préciser ce que vous apprendrez en lisant cet ouvrage. En revanche, voici ce que j'ai l'intention de vous enseigner :

- Ce que sont les algorithmes informatiques, une façon de les décrire et comment les évaluer.
- Des méthodes simples qui permettent de rechercher des informations sur un ordinateur.
- Des méthodes qui permettent de réorganiser les informations afin qu'elles apparaissent dans l'ordre souhaité (autrement dit les trier).
- Comment résoudre des problèmes de base dont la modélisation informatique se fait à l'aide d'une structure mathématique appelée **graphe**. Parmi leurs nombreuses applications, les graphes conviennent parfaitement à la modélisation des réseaux routiers (quelles intersections mènent à d'autres intersections par des chemins directs et quelles sont les longueurs de ces routes ?), des dépendances entre des tâches (quelle tâche doit venir avant les autres ?), des relations financières (quels sont les taux de change entre les différentes monnaies ?) ou des interactions entre des personnes (qui connaît qui ? qui n'aime pas qui ? quel acteur apparaît dans un film, au côté de quel autre ?).
- Comment résoudre des problèmes qui posent des questions sur des chaînes de texte. Certains de ces problèmes se rencontrent dans différents domaines, comme la biologie où les caractères représentent des molécules de base et les chaînes de caractères, une structure ADN.
- Les principes de base de la cryptographie. Même si vous n'avez jamais chiffré un message vous-même, il est probable que votre ordinateur l'ait déjà fait (par exemple lors de l'achat de produits en ligne).
- Les concepts fondamentaux de la compression de données.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- Comprendre que certains problèmes sont difficiles à résoudre sur un ordinateur en un temps raisonnable, ou, tout au moins, que personne ne sait encore comment procéder.

## Prérequis

Je l'ai indiqué précédemment, ce livre comprend du contenu mathématique. Si cela vous effraie, vous pouvez essayer de sauter ces passages ou de vous tourner vers un ouvrage moins technique. Toutefois, j'ai fait mon possible pour que ce contenu reste accessible au plus réfractaire.

Je ne suppose pas que vous ayez déjà écrit ou même lu un programme informatique. Si vous êtes capable de suivre des instructions données sous forme d'un plan, vous serez en mesure de comprendre la façon dont j'exprime les étapes qui constituent un algorithme. Si vous comprenez la blague suivante, vous avez déjà fait un grand pas :

> *Avez-vous déjà entendu parler de cet informaticien qui était resté bloqué dans sa douche ? Il se lavait les cheveux en suivant les instructions inscrites sur la bouteille de shampooing : « savonnez, rincez, recommencez ».*

Dans cet ouvrage, j'emploie un style relativement informel en espérant qu'une approche personnelle rendra le contenu plus accessible. Certains chapitres se fondent sur les précédents, mais les dépendances sont rares. Quelques-uns débutent de manière non technique, mais le deviennent progressivement. Si vous trouvez qu'un chapitre devient trop compliqué, vous aurez probablement intérêt à lire au moins le début du suivant.

Copyright © 2013 Dunod.

## Erreurs

Si vous découvrez une erreur dans cet ouvrage, n'hésitez pas à m'en avertir par courrier électronique à l'adresse *unlocked@mit.edu*.

## Remerciements

Je suis extrêmement reconnaissant à mes co-auteurs de l'ouvrage *Algorithmique*, Charles Leiserson, Ron Rivest et Cliff Stein, car une grande partie du présent ouvrage se fonde sur son contenu. Vous constaterez que j'y fais souvent effrontément référence (comprendre que j'en prends des extraits). En rédigeant cet ouvrage seul, j'ai réalisé combien la collaboration avec Charles, Ron et Cliff pouvait me manquer. Je remercie également toutes les personnes mentionnées dans l'avant-propos du CLRS (sigle utilisé dans

le monde entier pour désigner l'ouvrage *Algorithmique*, en référence aux initiales de ses quatre auteurs).

J'ai également fait appel au contenu des cours que je donne à Dartmouth, notamment ceux intitulés *Computer Science* 1, 5 et 25. Merci à mes étudiants dont les questions pertinentes m'ont permis de connaître les approches pédagogiques qui fonctionnent et dont les silences glaciaux ont révélé celles à éviter.

La rédaction de ce livre a été suggérée par Ada Brunstein, notre éditrice chez MIT Press, pendant que nous préparions la troisième édition du CLRS. Ada a depuis quitté la société et a été remplacée par Jim DeWolf. À l'origine cet ouvrage devait rejoindre la collection *Essential Knowledge*, mais il a semblé trop technique (pouvez-vous imaginer cela, j'ai écrit un livre trop technique pour le MIT !). Jim a parfaitement géré cette situation potentiellement délicate en me permettant d'écrire non pas l'ouvrage imaginé à l'origine par MIT Press mais celui que je souhaitais. J'ai également apprécié le soutien d'Ellen Faran et de Gita Devi Manaktala, toutes deux œuvrant chez MIT Press.

Julie Sussman s'était chargée de la révision technique de la deuxième et de la troisième édition du CLRS. Je suis vraiment ravi qu'elle ait pu également s'occuper de celui-ci. Elle n'a rien laissé passer. Pour preuve, voici une partie du message que Julie m'a envoyé à propos d'un premier brouillon du Chapitre 5 :

> *« Cher M. Cormen,*
>
> *Les autorités ont appréhendé un chapitre évadé qui s'était caché dans votre livre. Nous ne sommes pas en mesure de déterminer de quel ouvrage il s'était échappé, mais nous ne pouvons pas imaginer qu'il ait pu se glisser dans le vôtre pendant autant de mois sans que vous le sachiez. Nous n'avons donc pas d'autre choix que de vous tenir pour responsable. Nous espérons que vous vous chargerez de le remanier et d'en faire un bon citoyen de votre livre. Vous trouverez ci-joint le rapport rédigé par l'officier qui a procédé à l'arrestation, Julie Sussman. »*

Je ne suis pas un spécialiste de la cryptographie et le chapitre qui traite de ses principes de base a énormément bénéficié des commentaires et des suggestions de Ron Rivest, Sean Smith, Rachel Miller et Huijia Rachel Lin. Une note de bas de page fait référence aux signes utilisés dans les matches de base-ball. Je remercie Bob Whalen, l'entraîneur du club de Dartmouth, pour m'avoir patiemment expliqué certains des systèmes employés. Ilana Arbisser a vérifié que les spécialistes de bio-informatique alignent les séquences ADN de la manière dont je l'explique au Chapitre 8. Travailler au département informatique du Dartmouth College, c'est vraiment génial. Mes collègues sont brillants et fraternels, et notre équipe professionnelle

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

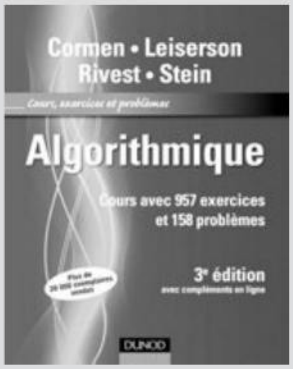
est à nulle autre pareille. Si vous recherchez un cours d'informatique de niveau premier ou second cycle, ou si vous recherchez un poste en informatique dans une université américaine, n'hésitez pas à contacter Dartmouth.

Merci enfin à mon épouse, Nicole Cormen, à mes parents, Renee et Perry Cormen, à ma sœur, Jane Maslin, et aux parents de Nicole, Colette et Paul Sage, pour leur amour et leur soutien. Mon père est certain que le dessin en page 2 est non pas un S mais un 5.

Tom Cormen, Hanover,
New Hampshire, novembre 2012

## Note de l'éditeur

Nous remercions M. Christophe Picouleau, professeur au Cnam de Paris, pour sa relecture critique de la traduction du chapitre 10 sur la complexité.




L'ouvrage [CLRS09] auquel Thomas Cormen fait souvent allusion dans cet ouvrage est son cours de référence dont il existe une version française :
*Algorithmique*
*Cours avec 957 exercices et 158 problèmes*
T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein,
3e édition, 1296 pages,
2010, Dunod.

Copyright © 2013 Dunod.

# 1 INTRODUCTION AUX ALGORITHMES

Commençons par une question récurrente : « *Qu'est-ce qu'un algorithme ?* »[1]

Une réponse générale pourrait être « un ensemble d'étapes qui permettent d'accomplir une tâche ». Dans notre vie quotidienne, nous exécutons de nombreux algorithmes, par exemple lorsque nous nous brossons les dents : ouvrir le tube dentifrice, prendre la brosse à dents, déposer du dentifrice sur la brosse, fermer le tube, placer la brosse dans la bouche, brosser les dents verticalement pendant $N$ secondes, etc. Pour nous rendre sur notre lieu de travail, nous déroulons également un algorithme. Il en va ainsi pour toutes les tâches.

Cet ouvrage concerne toutefois des algorithmes qui s'exécutent sur des ordinateurs ou, de façon plus générale, sur des appareils informatiques. Tout comme les algorithmes que nous déroulons affectent votre vie quotidienne, ceux qui s'exécutent sur les ordinateurs affectent ceux-ci. Lorsque nous utilisons notre GPS pour connaître l'itinéraire vers notre destination, il exécute un algorithme du « plus court chemin » pour la déterminer. Lorsque nous achetons des produits sur Internet, nous utilisons (ou devrions utiliser) un site web sécurisé qui met en place un algorithme de chiffrement. Ces produits sont apportés à notre domicile par un service de livraison qui utilise des algorithmes afin de répartir les paquets dans les différents camions et de déterminer l'ordre de passage chez les clients. Les algorithmes sont partout : sur notre ordinateur portable, sur les serveurs, sur notre smartphone, sur les systèmes embarqués (notre voiture, notre micro-onde ou notre climatisation), etc.

Quelle est la différence entre un algorithme qui s'exécute sur un ordinateur et un algorithme que nous déroulons ? Contrairement à un ordinateur, nous pouvons accepter que l'algorithme soit décrit de façon peu précise.

1. Ou comme le demanderait un collègue avec qui j'avais l'habitude de jouer au hockey : *« Qu'est-ce qu'un nalgorithme ? »*

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Par exemple, si nous prenons notre voiture pour nous rendre sur notre lieu de travail, l'algorithme pourrait stipuler « si la circulation est encombrée, changer de chemin ». Nous saurons sans doute interpréter « mauvaise circulation », pas un ordinateur.

Par conséquent, un algorithme informatique est un ensemble d'étapes qui permettent d'accomplir une tâche et qui est décrit de manière suffisamment précise pour qu'un ordinateur puisse l'exécuter. Si vous avez déjà écrit des programmes informatiques en Java, C, C++, Python, Fortran, Matlab ou autres, vous savez ce que signifie précision. Dans le cas contraire, vous le découvrirez rapidement en étudiant les algorithmes étudiés dans cet ouvrage.

Passons à la question suivante : « *Qu'attendons-nous d'un algorithme informatique ?* »

Les algorithmes informatiques résolvent des problèmes informatiques. Nous en attendons deux choses : proposer une solution juste à partir des données d'entrée du problème et utiliser pour cela les ressources de calcul de manière efficace. Examinons tour à tour ces deux points.

## Exactitude

Que signifie *« proposer une solution exacte à un problème »* ? En général, nous sommes en mesure d'indiquer précisément en quoi consiste une solution exacte. Par exemple, une solution juste donnée par notre GPS à notre problème de recherche du meilleur trajet vers notre destination serait la route qui, parmi toutes les possibilités existantes entre notre point de départ et notre point d'arrivée, prend le moins de temps. Ou bien celle qui correspond à la plus courte distance. Ou bien celle qui prend le moins de temps en évitant les péages. Bien entendu, les informations utilisées par le GPS pour déterminer l'itinéraire peuvent ne pas correspondre à la réalité. À moins qu'il ne soit capable d'accéder aux informations de trafic en temps réel, il peut supposer que le temps de parcours est égal à la distance divisée par les limitations de vitesse. Cependant, en cas d'embouteillage, le GPS pourrait donner un mauvais conseil si l'objectif était le trajet le plus rapide. Néanmoins, l'algorithme d'établissement de l'itinéraire exécuté par le GPS reste correct même si les entrées ne le sont pas. En effet, à partir des informations fournies à l'algorithme, il renvoie le chemin le plus rapide.

Dans le cas de certains problèmes, il peut être difficile, voire impossible, d'indiquer si un algorithme produit une solution exacte. Prenons pour exemple un algorithme de reconnaissance de caractères. L'image de $11 \times 6$ pixels suivante correspond-elle à un 5 ou à un S ?

Copyright © 2013 Dunod.

Pour certaines personnes, il s'agit d'un 5, mais, pour d'autres, d'un S. Dans ce cas, comment pouvons-nous déclarer que le résultat donné par l'ordinateur est exact ou non ? Dans cet ouvrage, nous nous focalisons sur les algorithmes qui produisent des solutions connaissables.

Nous pouvons parfois accepter qu'un algorithme informatique produise une réponse incorrecte, tant que nous contrôlons leur fréquence. Le chiffrement nous en donne un bon exemple. Le cryptosystème RSA très utilisé détermine si des nombres très grands (vraiment très grands, avec plusieurs centaines de chiffres) sont premiers. Si vous avez déjà écrit un programme informatique, vous serez probablement capable d'en développer un qui détermine si le nombre $n$ est premier. Il suffit de tester tous les diviseurs potentiels, de 2 à $n - 1$. Si l'un d'eux est un diviseur de $n$, alors $n$ est un nombre composé. Dans le cas contraire, $n$ est un nombre premier. Cependant, si $n$ comprend plusieurs centaines de chiffres, les diviseurs potentiels sont extrêmement nombreux et, même sur un ordinateur très rapide, les tests ne pourront pas se faire en un temps raisonnable. Nous pouvons évidemment procéder à quelques optimisations, comme supprimer tous les nombres pairs après avoir déterminé que 2 n'est pas un diviseur ou arrêter le test en arrivant à $\sqrt{n}$ (en effet, si $d$ est supérieur à $\sqrt{n}$ et si $d$ est un diviseur de $n$, alors $n / d$ est inférieur à $\sqrt{n}$ et est également un diviseur de $n$ ; par conséquent, si $n$ possède un diviseur, il aura été trouvé avant d'arriver à $\sqrt{n}$). Si $n$ est constitué de centaines de chiffres, alors $\sqrt{n}$ en comprendra environ moitié moins, mais cela reste encore un très grand nombre. Voici une bonne nouvelle : il existe un algorithme qui détermine rapidement si un nombre est premier. Voici la mauvaise nouvelle : il peut se tromper. Plus précisément, s'il déclare que $n$ est un nombre composé, alors c'est effectivement le cas. En revanche, s'il déclare que $n$ est premier, il existe une possibilité que $n$ soit en réalité composé. La mauvaise nouvelle ne l'est pourtant pas autant qu'on pourrait le penser : nous pouvons faire en sorte que le taux d'erreur soit réellement faible, par exemple de l'ordre d'une sur $2^{50}$. Pour la plupart d'entre nous, ce taux est suffisamment faible pour que nous puissions employer dans RSA cette façon de déterminer si un nombre est premier.

Pour une autre classe d'algorithmes, appelés *algorithmes d'approximation*, l'exactitude est un point plus délicat. Ces algorithmes s'appliquent aux problèmes d'optimisation dans lesquels nous souhaitons obtenir la meilleure solution en fonction d'une certaine mesure quantitative. Trouver l'itinéraire le plus rapide, comme le fait un GPS, en est un exemple ; dans ce cas, la mesure quantitative est la durée du voyage. Pour certains problèmes, aucun algorithme ne permet d'arriver à une solution optimale en un temps raisonnable, mais nous savons qu'un algorithme d'approximation permet d'obtenir, en un temps raisonnable, une solution quasi optimale. En général, « quasi optimale » signifie que la mesure quantitative de la

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

solution déterminée par l'algorithme d'approximation est éloignée d'un facteur connu de la mesure quantitative de la solution optimale. Tant que nous pouvons préciser ce facteur, nous pouvons dire qu'une solution correcte fournie par l'algorithme d'approximation correspond à toute solution qui s'éloigne au plus de ce facteur de la solution optimale.

## Utilisation des ressources

Dans le contexte d'un algorithme, que signifie une utilisation efficace des ressources de calcul ? Dans notre présentation des algorithmes d'approximation, nous avons fait allusion à une mesure d'efficacité : le temps. Un algorithme qui produit une solution correcte mais en prenant pour cela beaucoup de temps aura probablement un intérêt réduit, voire nul. S'il faut une heure à notre GPS pour déterminer le trajet, allons-nous vraiment nous en servir ? Le temps est ainsi la première mesure d'efficacité utilisée pour évaluer un algorithme, après que nous avons montré qu'il produit une solution exacte. Mais il existe d'autres mesures. Nous pouvons être concernés par la quantité de mémoire nécessaire à l'exécution de l'algorithme (son « empreinte mémoire ») car elle doit s'effectuer dans l'espace mémoire disponible. Voici d'autres ressources dont un algorithme peut avoir besoin : communication réseau, données aléatoires (les algorithmes qui effectuent des choix aléatoires ont besoin d'une source de nombres aléatoires) ou opérations disque (pour les algorithmes conçus de façon à manipuler des données stockées sur le disque).

Dans cet ouvrage, comme dans la plupart des livres qui traitent des algorithmes, nous nous focaliserons sur une seule ressource : le temps. Comment pouvons-nous juger du temps requis par un algorithme ? Contrairement à l'exactitude, qui ne dépend pas de l'ordinateur sur lequel s'exécute l'algorithme, le temps d'exécution réel dépend de plusieurs facteurs externes à l'algorithme lui-même : la rapidité de l'ordinateur, le langage de programmation utilisé pour sa mise en œuvre, le compilateur ou l'interpréteur qui traduit le programme en code exécutable sur la machine, les compétences du programmeur qui écrit le programme et les autres activités qui ont lieu sur l'ordinateur en même temps que le programme. Par ailleurs, tout cela suppose que l'algorithme s'exécute sur un seul ordinateur et que toutes ses données soient présentes en mémoire.

Si nous évaluons la rapidité d'un algorithme en l'implémentant à l'aide d'un langage de programmation réel, en l'exécutant sur un ordinateur précis avec des entrées données et en mesurant le temps nécessaire à son exécution, nous ne savons pas quelle pourra être sa rapidité avec des entrées de taille différente ou avec des entrées différentes de même taille. Par ailleurs, si nous voulons comparer la vitesse de cet algorithme à celle d'un autre algorithme pour le même problème, nous devons les implémenter tous les

Copyright © 2013 Dunod.

deux et les exécuter avec des entrées différentes de différentes tailles. En résumé, comment pouvons-nous évaluer la rapidité d'un algorithme ?

La réponse se fonde sur une combinaison de deux idées. Premièrement, nous déterminons la durée d'un algorithme en fonction de la taille de son entrée. Dans notre exemple de recherche d'itinéraire, l'entrée correspond à une représentation de la carte routière et la taille dépend du nombre d'intersections et du nombre de routes qui les relient. (La taille physique du réseau routier n'a pas d'importance car nous pouvons représenter toutes les distances par des nombres, qui occupent tous la même taille dans l'entrée ; la longueur d'une route n'a pas d'impact sur la taille de l'entrée.) Dans un exemple plus simple, comme déterminer si un élément précis existe dans une liste donnée, la taille de l'entrée correspond au nombre d'éléments de cette liste.

Deuxièmement, nous nous focalisons sur la manière dont la rapidité de la fonction qui caractérise le temps d'exécution augmente avec la taille de l'entrée – le *taux de croissance* du temps d'exécution. Au Chapitre 2, nous donnerons les notations employées pour caractériser le temps d'exécution d'un algorithme. Le point le plus intéressant de notre approche est que nous examinons uniquement le terme dominant du temps d'exécution, sans tenir compte des coefficients. Autrement dit, nous nous concentrons sur l'**ordre de grandeur** du temps d'exécution. Par exemple, supposons que nous puissions déterminer qu'une certaine mise en œuvre d'un algorithme de recherche dans une liste de $n$ éléments prend $50n + 125$ cycles processeur. Le terme $50n$ domine le terme 125 dès que $n$ devient suffisamment grand, c'est-à-dire lorsque $n \geq 3$, et plus la liste est longue plus il est dominant. Par conséquent, dans la description du temps d'exécution de cet algorithme hypothétique, nous ne tenons pas compte du terme d'ordre inférieur 125. Vous pourriez être surpris d'apprendre que le coefficient 50 ne nous intéresse pas. Le temps d'exécution est donc caractérisé par une croissance linéaire en fonction de la taille de l'entrée, c'est-à-dire $n$. Prenons un autre exemple, celui d'un algorithme qui demande $20n^3 + 100n^2 + 300n + 200$ cycles processeur. Son temps d'exécution croît en fonction de $n^3$. À nouveau, les termes d'ordre inférieur – $100n^2$, $300n$ et 200 – deviennent de moins en moins significatifs au fur et à mesure que la taille de l'entrée $n$ augmente.

Dans la pratique, les coefficients que nous ignorons ont une importance. Toutefois, ils dépendent tellement de facteurs externes qu'il est possible que, lors de la comparaison des algorithmes A et B ayant le même ordre de grandeur et s'exécutant sur la même machine, l'algorithme A soit plus rapide que l'algorithme B pour une certaine combinaison de machine, de langage de programmation, de compilateur/interpréteur et de programmeur, alors qu'avec une autre combinaison B soit plus rapide que A. Bien entendu, si les deux algorithmes A et B produisent des solutions exactes et

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

si A s'exécute toujours deux fois plus rapidement que B, alors, toutes choses étant égales par ailleurs, nous choisirons A à la place de B. Cependant, du point de vue de la comparaison abstraite des algorithmes, nous nous intéressons à l'ordre de grandeur, en oubliant les coefficients et les termes d'ordre inférieur.

La réponse à la dernière question de ce chapitre, « *Pourquoi s'intéresser aux algorithmes informatiques ?* », dépend de qui vous êtes.

## Algorithmes pour les non-informaticiens

Même si vous ne vous considérez pas comme un informaticien, les algorithmes informatiques vous concernent. En effet, à moins que vous ne partiez à l'aventure dans une contrée sauvage sans un GPS, vous les utilisez probablement quotidiennement. Avez-vous effectué une recherche sur Internet aujourd'hui ? Le moteur de recherche que vous avez utilisé, qu'il s'agisse de Google, de Bing ou d'un autre, se fonde sur des algorithmes complexes pour parcourir le Web et décider de l'ordre dans lequel présenter les résultats. Avez-vous conduit votre voiture aujourd'hui ? À moins qu'il s'agisse d'un véhicule de collection, son ordinateur de bord a réalisé des millions de choix au cours de votre déplacement, tous déterminés par des algorithmes. Je pourrais citer de nombreux autres exemples.

En tant qu'utilisateur final des algorithmes, vous vous devez d'en savoir un peu plus sur leur conception, leur caractérisation et leur évaluation. Je suppose que cela vous intéresse au moins un minimum, car vous avez acheté cet ouvrage et l'avez lu jusqu'à ces lignes. Voyons si nous pouvons vous en donner plus afin que vous puissiez participer aux conversations lors de votre prochain cocktail si jamais les algorithmes venaient à en être le sujet[1].

## Algorithmes pour les informaticiens

Si vous êtes informaticien, il est préférable que vous vous intéressiez de près aux algorithmes informatiques. Ils sont non seulement au cœur de tout ce qui se trouve dans l'ordinateur, mais ils constituent également autant une technologie que tous ces composants. Vous pouvez acheter un ordinateur au prix fort, avec le dernier et le plus puissant des processeurs, mais vous

1. Je sais parfaitement que, à moins de résider dans la Silicon Valley, il est peu probable que les algorithmes soient un sujet débattu lors des cocktails auxquels vous participez. Cependant, quelle qu'en soit la raison, nous, professeurs d'informatique, pensons qu'il est important que nos étudiants ne nous mettent pas dans l'embarras en raison d'un manque de connaissances dans certains domaines de l'informatique.

Copyright © 2013 Dunod.

devrez exécuter de bons algorithmes correctement implémentés pour que votre argent ait été dépensé intelligemment.

Voici un exemple qui illustre combien les algorithmes sont une technologie. Au Chapitre 3, nous en étudierons plusieurs qui trient une liste de $n$ valeurs par ordre croissant. Certains d'entre eux ont des temps d'exécution qui augmentent selon $n^2$, alors que pour d'autres cette croissance est seulement fonction de $n \lg n$. Que signifie $\lg n$ ? Il s'agit du logarithme en base 2 de $n$, c'est-à-dire $\log_2 n$. Les informaticiens se servent si souvent des logarithmes en base 2 qu'à l'instar des mathématiciens et des scientifiques qui utilisent le raccourci $\ln n$ pour le logarithme népérien ($\log_e n$), ils ont leur propre raccourci. Puisque la fonction $\lg n$ est l'inverse d'une fonction exponentielle, elle croît très lentement avec $n$. Si $n = 2^x$, alors $x = \lg n$. Par exemple, $2^{10} = 1\,024$ et donc $\lg 1\,024$ vaut seulement 10. De même, $2^{20} = 1\,048\,576$ et $\lg 1\,048\,576$ vaut uniquement 20 ; $2^{30} = 1\,073\,741\,824$ et $\lg 1\,073\,741\,824$ vaut donc 30. Par conséquent, une croissance de $n \lg n$ comparée à une croissance de $n^2$ échange un facteur $n$ contre un facteur de seulement $\lg n$. C'est plutôt une bonne affaire !

Voyons cet exemple de façon plus concrète en laissant un ordinateur rapide (l'ordinateur A) exécuter un algorithme de tri dont le temps d'exécution sur $n$ valeurs croît en $n^2$, et en laissant un ordinateur lent (l'ordinateur B) exécuter un algorithme de tri dont le temps d'exécution croît en $n \lg n$. Le tri va s'effectuer sur un tableau de 10 millions d'entrées (10 millions de nombres pourraient vous sembler important mais, s'il s'agit de valeurs sur 8 octets, l'entrée occupe environ 80 Mo, ce qui tient largement dans la mémoire des ordinateurs actuels, même ceux d'entrée de gamme). Supposons que l'ordinateur A soit capable d'exécuter 10 milliards d'instructions par seconde (plus rapide que tout ordinateur séquentiel disponible au moment de l'écriture de ces lignes) et que l'ordinateur B n'exécute que 10 millions d'instructions par seconde. En puissance brute, A est donc 1 000 fois plus rapide que B. Pour que la différence soit encore plus manifeste, supposons que le meilleur développeur au monde ait implémenté en langage machine l'algorithme pour l'ordinateur A et que le code résultant nécessite $2n^2$ instructions pour trier $n$ nombres. Dans le cas de l'ordinateur B, le programmeur est moins performant et a utilisé un langage de haut niveau avec un compilateur inefficace. Le code résultant demande $50\, n \lg n$ instructions. Pour trier 10 millions de nombres, voici le temps requis par l'ordinateur A :

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions / secondes}} = 20\,000 \text{ secondes}$$

Il faut donc plus de 5,5 heures. Voici le même calcul pour l'ordinateur B :

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions / secondes}} \approx 1\,163 \text{ secondes}$$

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Cela correspond à moins de 20 minutes. Autrement dit, en utilisant un algorithme dont le temps d'exécution augmente plus lentement, l'ordinateur B, même avec un compilateur médiocre, exécute le code environ 17 fois plus rapidement que l'ordinateur A ! L'intérêt d'un algorithme en $n \lg n$ est encore plus prononcé lors du tri de 100 millions de nombres : l'algorithme en $n^2$ sur l'ordinateur A demande plus de 23 jours, tandis que l'algorithme en $n \lg n$ sur l'ordinateur B prend moins de 4 heures. De façon générale, plus la taille du problème augmente, plus l'intérêt relatif d'un algorithme en $n \lg n$ augmente.

Malgré les améliorations impressionnantes que nous voyons au niveau du matériel des ordinateurs, les performances d'un système global dépendent autant du choix d'algorithmes efficaces que de composants rapides ou d'un système d'exploitation efficace. Les avancées en matière de technologies informatiques sont tout aussi étonnantes que celles réalisées au niveau des algorithmes.

## Lectures complémentaires

À mon avis, nécessairement partial, l'information la plus claire et la plus utile concernant les algorithmes informatiques se trouve dans l'ouvrage *Algorithmique* [CLRS09]. Ce livre est souvent appelé « CLRS », d'après les initiales des auteurs. Je l'ai largement exploité pour le contenu du présent ouvrage. Il est beaucoup plus complet que celui-ci, mais il suppose que vous ayez des connaissances en programmation informatique et il ne fait pas l'impasse sur les mathématiques. Si vous vous sentez à l'aise avec le niveau de mathématiques de cet ouvrage et si vous êtes prêt à approfondir le sujet, vous ne trouverez pas mieux que le CLRS (à mon humble avis, évidemment).

L'ouvrage *Nine Algorithms That Changed the Future* par John MacCormick [Mac12] décrit plusieurs algorithmes et les aspects connexes de l'informatique qui affectent votre vie quotidienne. Son approche est moins technique que celle employée dans le présent ouvrage. Si celle-ci vous semble trop mathématique, je vous conseille la lecture du livre de MacCormick. Vous devriez pouvoir suivre son contenu même si vos connaissances en mathématiques sont maigres.

Bien qu'improbable, si vous trouvez que le CLRS est trop édulcoré, vous pouvez vous tourner vers la collection *The Art of Computer Programming* par Donald Knuth [Knu97, Knu98a, Knu98b, Knu11]. Bien que l'intitulé puisse laisser croire que ces ouvrages se focalisent sur la manière d'écrire du code, ils proposent en réalité de brillantes analyses des algorithmes. Sachez toutefois que leur contenu est dense. Si vous vous interrogez sur les origines

Copyright © 2013 Dunod.

du mot algorithme, Knuth précise qu'il vient du nom « Al-Khawarizmi » d'un mathématicien perse qui vivait au IX$^{e}$ siècle.

Outre le CLRS, plusieurs autres excellents ouvrages sur les algorithmes informatiques ont été publiés au fil des ans. Les notes du Chapitre 1 du CLRS en recensent de nombreux. Plutôt que de reproduire cette liste ici, je préfère que vous consultiez le CLRS.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# 2 Décrire et évaluer des algorithmes

Au chapitre précédent, nous avons expliqué que nous exprimons le temps d'exécution d'un algorithme informatique comme une fonction de la taille de l'entrée, en nous intéressant plus particulièrement à l'ordre de grandeur de ce temps. Dans ce chapitre, nous allons expliquer comment décrire des algorithmes informatiques, puis nous présenterons les notations utilisées pour caractériser leur temps d'exécution. Nous terminerons par l'étude de quelques techniques utilisées pour concevoir et comprendre des algorithmes.

## Décrire des algorithmes informatiques

Il est toujours possible de décrire un algorithme informatique sous forme d'un programme exécutable écrit dans l'un des langages de programmation répandus, comme Java, C, C++, Python ou Fortran. C'est l'approche que prennent certains ouvrages d'algorithmique. Cependant, l'utilisation de langages de programmation réels pour spécifier des algorithmes a l'inconvénient d'exposer les détails du langage et de masquer les idées sous-jacentes aux algorithmes. Une autre approche, retenue dans l'ouvrage *Algorithmique*, consiste à utiliser du pseudocode, qui ressemble à un mélange entre des langages de programmation divers et du texte en français. Si vous avez déjà employé un langage de programmation, la lecture d'un pseudocode ne vous posera aucune difficulté. En revanche, si vous n'avez jamais programmé, il risque de vous sembler un tantinet mystérieux.

Dans ce livre, je n'essaie pas de décrire des algorithmes en ciblant un logiciel ou un matériel, mais la matière grise qui se trouve entre vos oreilles. Puisque je suppose également que vous n'avez jamais écrit un programme informatique, les algorithmes ne sont pas exprimés à l'aide d'un langage de programmation réel ni même avec du pseudocode. À la place, j'utilise le français, en faisant autant que possible une analogie avec le monde réel. Pour indiquer ce qu'il se passe à un moment donné (ce que nous appelons « flux de contrôle » en programmation), j'utiliserai des listes et des listes

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

imbriquées. Si vous voulez implémenter un algorithme dans un langage de programmation réel, je vous rendrai justice pour avoir été capable de traduire ma description en un code exécutable.

Même si je vais essayer de proposer des descriptions aussi peu techniques que possible, cet ouvrage traite d'algorithmes pour les ordinateurs et une terminologie informatique est donc incontournable. Par exemple, les programmes sont faits de *procédures* (également appelées fonctions ou méthodes dans les langages de programmation), qui précisent comment réaliser des tâches. Pour qu'une procédure réalise les traitements qu'elle est supposée faire, nous devons l'*appeler*. Lors de l'appel d'une procédure, nous fournissons en entrée des données (habituellement au moins une, mais certaines procédures n'en prennent aucune). Les données sont définies sous forme de *paramètres* placés entre des parenthèses après le nom de la procédure. Par exemple, pour calculer la racine carrée d'un nombre, nous pouvons définir la procédure RACINE-CARRÉE($x$). Dans ce cas, la donnée passée à la procédure est représentée par le paramètre $x$. L'appel d'une procédure peut produire ou non une sortie en fonction de la façon dont elle a été définie. Lorsqu'*elle produit* une sortie, nous considérons généralement que celle-ci correspond à des informations renvoyées à l'appelant. Nous disons que la procédure *retourne* une valeur.

De nombreux programmes et algorithmes manipulent des tableaux de données. Un *tableau* regroupe des données de même type au sein d'une seule entité. Vous pouvez voir un tableau comme une table qui, à partir de l'*indice* d'une *entrée*, permet de faire référence à l'élément qui se trouve à cet indice dans le tableau. Voici par exemple une table des cinq premiers présidents de la V$^{e}$ République française :

| Indice | Président |
|---|---|
| 1 | Charles de Gaulle |
| 2 | Georges Pompidou |
| 3 | Valéry Giscard d'Estaing |
| 4 | François Mitterrand |
| 5 | Jacques Chirac |

Dans cette table, l'élément à l'indice 4 est François Mitterrand. Nous la voyons non pas comme cinq entités séparées, mais comme une table constituée de cinq entrées. Un tableau est comparable. Ses indices sont des nombres consécutifs dont la valeur initiale est quelconque, mais elle est en général égale à 1[1]. Pour désigner un élément précis d'un tableau, nous

1. Si vous programmez en Java, C ou C++, vous avez l'habitude d'utiliser des tableaux dont l'indice débute à 0. Si cela convient parfaitement aux ordinateurs, les humains préfèrent souvent une valeur de départ à 1.

Copyright © 2013 Dunod.

indiquons le nom de celui-ci et l'indice de l'élément correspondant entre crochets. Par exemple, voici comment faire référence à l'élément d'indice $i$ dans le tableau $A$ : $A[i]$.

Les tableaux possèdent une autre caractéristique importante : il faut le même temps pour accéder à n'importe quel élément d'un tableau. Après que nous avons donné à l'ordinateur l'indice $i$ dans un tableau, l'accès à l'élément correspondant se fait aussi rapidement qu'au premier, quelle que soit la valeur de $i$.

Étudions à présent notre premier algorithme : rechercher une valeur précise dans un tableau. Autrement dit, nous avons un tableau et voulons savoir si une de ses entrées contient la valeur indiquée. Pour comprendre comment se passe la recherche dans un tableau, considérons-le comme une longue étagère remplie de livres et supposons que nous souhaitions savoir si l'un d'eux a été écrit par Jonathan Swift. Les ouvrages peuvent être organisés de différentes manières, par exemple par ordre alphabétique des auteurs, par ordre alphabétique des titres ou, dans une bibliothèque, par références. Il est également possible qu'ils soient, comme dans mon cas, non classés.

Si nous ne pouvons pas supposer que les ouvrages sont classés sur l'étagère, comment pouvons-nous en trouver un écrit par Jonathan Swift ? Il suffit de dérouler l'algorithme suivant. Nous partons de l'extrémité gauche de l'étagère et nous examinons l'ouvrage qui se trouve à cette place. S'il a été écrit par Swift, nous avons trouvé le livre. Sinon nous examinons l'ouvrage suivant placé à sa droite et, s'il a été écrit par Swift, nous avons trouvé le livre. Sinon nous continuons avec l'ouvrage situé à droite, en examinant livre après livre, jusqu'à en trouver un écrit par Swift ou à atteindre l'extrémité droite de l'étagère. Dans ce cas, nous pouvons en conclure qu'elle ne contient aucun ouvrage de Jonathan Swift. (Au Chapitre 3, nous verrons comment effectuer cette recherche avec des livres classés.)

Voyons à présent comment décrire ce problème de recherche en termes informatiques. Considérons que les livres sur l'étagère forment un tableau d'ouvrages. Celui situé à l'extrémité gauche est en position 1, le suivant à sa droite est en position 2, etc. Si l'étagère comprend $n$ livres, alors celui à l'extrême droite est en position $n$. Nous souhaitons déterminer la position sur l'étagère de n'importe quel livre écrit par Jonathan Swift.

En tant que problème informatique général, nous recevons un tableau $A$ (l'intégralité de l'étagère remplie d'ouvrages) de $n$ éléments (les ouvrages individuels) et nous souhaitons déterminer si une valeur $x$ (un livre écrit par Jonathan Swift) est présente dans $A$. Dans l'affirmative, nous voulons obtenir l'indice $i$ tel que $A[i] = x$ (nous trouvons un livre de Jonathan Swift sur l'étagère à la position $i$). Nous devons également trouver une manière d'indiquer que le tableau $A$ ne contient pas $x$ (aucun livre présent sur l'étagère n'a été écrit par Jonathan Swift). Nous ne supposons pas que $x$ n'apparaît

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

qu'une seule fois dans le tableau (nous pouvons avoir plusieurs exemplaires du même ouvrage). Nous voulons que l'algorithme de recherche indique tout indice auquel nous pouvons trouver $x$ dans le tableau. Nous supposons que les indices débutent à la valeur 1 et que les emplacements des éléments vont donc de $A[1]$ à $A[n]$.

Si nous recherchons n'importe quel ouvrage de Jonathan Swift en commençant par l'extrémité gauche de l'étagère, en vérifiant les ouvrages un par un et en progressant vers la droite, nous mettons en œuvre une technique appelée **recherche linéaire**. D'un point de vue informatique, nous commençons au début du tableau, examinons chacun de ses éléments ( $A[1]$, puis $A[2]$, puis $A[3]$ et ainsi de suite jusqu'à $A[n]$) et enregistrons l'emplacement de $x$ si nous le trouvons.

La procédure suivante, nommée RECHERCHE-LINÉAIRE, prend trois paramètres, que nous séparons par des virgules dans la spécification.

*Procédure* RECHERCHE-LINÉAIRE($A$,$n$,$x$)

*Entrées :*

- $A$ : un tableau.
- $n$ : le nombre d'éléments de $A$ à examiner.
- $x$ : la valeur recherchée.

*Sortie :* Soit l'indice $i$ pour lequel $A[i] = x$, soit la valeur spéciale NON-TROUVÉ, qui peut correspondre à n'importe quel indice invalide pour le tableau, comme 0 ou un entier négatif.

1. Fixer *réponse* à NON-TROUVÉ.
2. Pour chaque indice $i$, en allant de 1 à $n$, dans l'ordre :
   A. Si $A[i] = x$, alors fixer *réponse* à la valeur de $i$.
3. Retourner la valeur de *réponse* en tant que sortie.

Outre les paramètres $A$, $n$ et $x$, la procédure RECHERCHE-LINÉAIRE utilise une **variable** nommée *réponse*. À l'étape 1, elle lui **affecte** la valeur initiale NON-TROUVÉ. À l'étape 2, elle examine chaque entrée du tableau, de $A[1]$ à $A[n]$, pour savoir si elle contient la valeur $x$. Lorsque l'entrée $A[i]$ est égale à $x$, l'étape 2A affecte la valeur courante de $i$ à *réponse*. Si $x$ est présent dans le tableau, alors la valeur de sortie retournée lors de l'étape 3 est le dernier indice auquel $x$ est apparu. Si $x$ est absent du tableau, alors le test d'égalité effectué à l'étape 2A n'est jamais vrai et la valeur de sortie retournée est NON-TROUVÉ (elle avait été affectée à *réponse* lors de l'étape 1).

Avant de poursuivre notre étude de la recherche linéaire, attardons-nous sur la manière de préciser la répétition d'actions, comme à l'étape 2. Dans les algorithmes, il est assez fréquent d'effectuer une action pour une

Copyright © 2013 Dunod.

variable qui doit prendre successivement toutes les valeurs d'une certaine plage. La répétition d'action s'appelle une **boucle** et chaque tour de boucle correspond à une **itération**. Dans le cas de la boucle de l'étape 2, il est écrit « Pour chaque indice $i$, en allant de 1 à $n$, dans l'ordre ». Désormais, je remplacerai cette formulation par « Pour $i = 1$ à $n$ ». Cette écriture est plus courte tout en transmettant la même structure. Lorsqu'une boucle est écrite de cette manière, nous devons donner à la **variable de boucle** (dans ce cas $i$) une valeur initiale (dans ce cas 1) et, dans chaque itération, nous devons comparer la valeur courante de cette variable à une limite (dans ce cas $n$). Si la valeur courante de la variable est inférieure ou égale à la limite, alors nous exécutons le *corps* de la boucle (dans ce cas l'étape 2A). Après que l'itération a exécuté le corps de la boucle, nous **incrémentons** la variable de boucle – nous lui ajoutons 1 – et reprenons en comparant cette variable, à présent avec sa nouvelle valeur, à la limite. Autrement dit, nous répétons les opérations suivantes jusqu'à ce que la variable dépasse la limite : comparer la variable de boucle à la limite, exécuter le corps de la boucle et incrémenter la variable de boucle. L'exécution se poursuit ensuite à l'étape qui vient immédiatement après le corps de la boucle (dans ce cas l'étape 3). Une boucle de la forme « Pour $i = 1$ à $n$ » réalise $n$ itérations et $n + 1$ comparaisons avec la limite (car la valeur de la variable de boucle dépasse la limite lors du $n + 1$-ème test).

J'espère que vous avez compris que la procédure RECHERCHE-LINÉAIRE retourne toujours une réponse exacte. Vous aurez peut-être également remarqué qu'elle n'est pas efficace : elle poursuit sa recherche dans le tableau même après avoir trouvé un indice $i$ pour lequel $A[i] = x$. Normalement, si vous trouvez l'ouvrage qui convient sur votre étagère, vous n'allez pas poursuivre votre recherche. Nous devons donc revoir notre procédure de recherche linéaire afin qu'elle stoppe son traitement dès qu'elle a trouvé la valeur $x$ dans le tableau. Nous supposons que lorsque nous lui demandons de retourner une valeur, la procédure renvoie celle-ci immédiatement à l'appelant, qui reprend le contrôle.

*Procédure* MEILLEURE-RECHERCHE-LINÉAIRE$(A, n, x)$
*Entrées et sortie :* identiques à celles de RECHERCHE-LINÉAIRE.
1. Pour $i = 1$ à $n$ :
   A. Si $A[i] = x$, alors retourner la valeur de $i$ comme sortie.
2. Retourner NON-TROUVÉ comme sortie.

Croyez-le ou non, nous pouvons rendre la recherche linéaire encore plus efficace. Vous le constatez, lors de chaque tour de boucle de l'étape 1, la procédure MEILLEURE-RECHERCHE-LINÉAIRE effectue deux tests. Le premier, à l'étape 1, détermine si $i \leq n$ (dans l'affirmative, une autre itération

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

de la boucle est réalisée). Le second test, une égalité, se trouve à l'étape 1A. Par analogie à une recherche sur une étagère, ces tests correspondent à deux contrôles pour chaque livre : sommes-nous arrivés à la fin de l'étagère et, si ce n'est pas le cas, le livre suivant est-il de Jonathan Swift ? Nous ne serons évidemment pas sanctionnés si nous dépassons l'extrémité de l'étagère (sauf si notre visage reste collé aux ouvrages et que nous nous cognons la tête contre le mur sur lequel s'appuie l'étagère), mais, dans un programme informatique, il est fortement déconseillé d'essayer d'accéder à des éléments qui se trouvent en dehors du tableau. Le programme peut alors se planter ou corrompre des données.

Il est possible d'écrire une procédure qui n'effectue qu'un seul contrôle pour chaque ouvrage examiné. Et si nous savions à coup sûr qu'un livre de Jonathan Swift se trouve sur l'étagère ? Dans ce cas, nous serions assurés de le trouver et il ne serait plus utile de vérifier le dépassement de la fin de l'étagère. Il suffirait de tester si chaque livre a été écrit par Swift.

Mais peut-être que nous avons prêté tous nos ouvrages de Jonathan Swift ou que nous pensions en posséder un alors que ce n'était pas le cas. Nous ne pouvons donc pas être certain qu'un de ses livres se trouve sur l'étagère. Voici ce que nous pouvons faire. Nous prenons une boîte vide de la taille d'un livre et écrivons sur sa tranche « *Les Voyages de Gulliver* par Jonathan Swift ». Nous remplaçons le livre situé à l'extrême droite par cette boîte. Ensuite, au cours de la recherche de gauche à droite sur l'étagère, nous vérifions uniquement si le livre concerné a été écrit par Swift. Nous pouvons omettre le test de la fin de l'étagère car nous savons que nous allons trouver quelque chose de Jonathan Swift. La seule question est de savoir si nous avons vraiment trouvé un livre de Swift ou s'il s'agit de la boîte vide que nous lui avons attribuée. Dans ce dernier cas, nous n'avons pas véritablement trouvé un ouvrage de Swift. Cependant, ce cas est facile à identifier et le test correspondant n'est effectué qu'une seule fois à la fin de la recherche plutôt que pour chaque ouvrage sur l'étagère.

Un détail ne vous aura peut-être pas échappé : et si le seul livre de Jonathan Swift que nous possédions sur l'étagère était celui le plus à droite ? Nous l'avons remplacé par la boîte vide et, lorsque la recherche se termine sur cette boîte, nous concluons que nous n'avions aucun ouvrage de cet auteur. Nous devons effectuer un contrôle supplémentaire pour cette possibilité, mais il s'agit non pas d'un test effectué pour chaque ouvrage sur l'étagère mais d'une seule vérification.

Du point de vue d'un algorithme informatique, nous allons placer la valeur de $x$ que nous recherchons en dernière position, $A[n]$, après avoir sauvegardé le contenu de $A[n]$ dans une autre variable. Si nous trouvons $x$, nous vérifions qu'il s'agit bien d'un livre. La valeur que nous plaçons dans le tableau est une **sentinelle**, mais vous pouvez la voir comme la boîte vide.

Copyright © 2013 Dunod.

*Procédure* RECHERCHE-LINÉAIRE-SENTINELLE($A$,$n$,$x$)

*Entrées et sortie :* identiques à celles de RECHERCHE-LINÉAIRE.

1. Sauvegarder $A[n]$ dans *dernier* et placer $x$ dans $A[n]$.
2. Fixer $i$ à 1.
3. Tant que $A[i] \neq x$, faire la chose suivante :
   A. Incrémenter $i$.
4. Restaurer $A[n]$ à partir de *dernier.*
5. Si $i < n$ ou $A[n] = x$, alors retourner la valeur de $i$ comme sortie.
6. Sinon retourner NON-TROUVÉ comme sortie.

L'étape 3 correspond à une boucle mais elle ne se fonde pas sur une variable de boucle. À la place, le corps de la boucle est exécuté tant qu'une condition reste vraie. Dans ce cas, la condition est $A[i] \neq x$. Voici comment fonctionne une telle boucle. Le test est effectué (dans ce cas $A[i] \neq x$) et, s'il est vrai, le corps de la boucle est exécuté (dans ce cas l'étape 3A, qui incrémente $i$). Le même déroulement reprend ensuite, avec l'exécution du corps de la boucle jusqu'à ce que le test devienne faux. Le contrôle passe ensuite à l'étape qui vient après le corps de la boucle (dans ce cas l'étape 4).

La procédure RECHERCHE-LINÉAIRE-SENTINELLE est un tantinet plus complexe que les deux précédentes. Puisqu'elle place $x$ dans $A[n]$ au cours de l'étape 1, nous sommes certains que $A[i]$ sera égal à $x$ pour l'un des tests de l'étape 3. À ce moment-là, nous sortons de la boucle et l'indice $i$ n'est plus modifié. Avant toute chose, l'étape 4 restaure la valeur d'origine dans $A[n]$. (Ma maman m'a toujours dit de remettre les choses à leur place lorsque j'avais fini de les utiliser.) Ensuite, nous devons déterminer si nous avons réellement trouvé $x$ dans le tableau. Puisque nous avons placé $x$ dans le dernier élément, $A[n]$, nous savons que si nous avons trouvé $x$ dans $A[i]$ avec $i < n$, alors notre recherche a été fructueuse et nous retournons $i$. Si nous avons trouvé $x$ dans $A[n]$, cela signifie qu'il n'était pas présent avant $A[n]$ et nous devons donc vérifier si $A[n]$ est égal à $x$. Dans l'affirmative, nous retournons l'indice $n$ qui, à ce stade, est égal à $i$. Dans le cas contraire, nous retournons NON-TROUVÉ. Tous ces tests sont effectués à l'étape 5, qui retourne l'indice approprié si $x$ se trouvait initialement dans le tableau. Si $x$ a été trouvé uniquement parce que nous l'avons placé dans le tableau à l'étape 1, alors l'étape 6 retourne NON-TROUVÉ. Bien que deux tests soient effectués après la fin de la boucle, la procédure RECHERCHE-LINÉAIRE-SENTINELLE n'en effectue qu'un seul à chaque itération de la boucle et se révèle donc plus efficace que RECHERCHE-LINÉAIRE et MEILLEURE-RECHERCHE-LINÉAIRE.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

## Caractériser les temps d'exécution

Revenons à la procédure Recherche-Linéaire de la page 14 et étudions son temps d'exécution. Rappelez-vous que nous voulons caractériser le temps d'exécution en fonction de la taille des entrées. Dans cet exemple, l'entrée est un tableau $A$ de $n$ éléments, ainsi que le nombre $n$ et la valeur $x$ recherchée. La taille de $n$ et celle de $x$ deviennent non significatives lorsque le tableau est volumineux. En effet, $n$ n'est qu'un entier et $x$ a la même taille que l'un des $n$ éléments du tableau. Nous disons donc que la taille de l'entrée est égale à $n$, c'est-à-dire le nombre d'éléments de $A$.

Nous devons faire quelques hypothèses simples sur la durée des opérations. Nous supposons que chaque opération individuelle, qu'il s'agisse d'une opération arithmétique (comme l'addition, la soustraction, la multiplication ou la division), d'une comparaison, d'une affectation de variable, d'une indexation dans un tableau, d'un appel de procédure ou du retour d'une valeur à partir d'une procédure, prend un temps figé, indépendant de la taille de l'entrée[1]. La durée peut varier en fonction des opérations (une division peut être plus longue qu'une addition), mais lorsqu'une étape comprend uniquement des opérations simples, chacune de ses exécutions se fait en un temps constant. Puisque les opérations exécutées diffèrent d'une étape à l'autre et en raison des facteurs externes recensés à la page 4, le temps d'exécution d'une étape peut varier selon les étapes. Nous supposons que chaque exécution de l'étape $i$ se fait en un temps $t_i$, où $t_i$ est une constante qui ne dépend pas de $n$.

Nous devons évidemment tenir compte du fait que certaines étapes sont effectuées à plusieurs reprises. Les étapes 1 et 3 s'exécutent une seule fois, mais qu'en est-il de l'étape 2 ? La comparaison de $i$ avec $n$ est effectuée $n + 1$ fois : $n$ fois alors que $i \leq n$ et une fois lorsque $i$ est égal à $n + 1$ et que la boucle est arrêtée. L'étape 2A s'exécute exactement à $n$ reprises, une fois pour chaque valeur de $i$ entre 1 et $n$. Nous ne savons pas à l'avance combien de fois la valeur de $i$ sera affectée à *réponse* ; ce nombre va de 0 (si $x$ est absent du tableau) à $n$ (si chaque élément du tableau est égal à $x$). Pour que notre comptabilité soit précise (en général elle ne le sera pas autant), nous devons reconnaître que l'étape 2 effectue deux traitements différents un nombre différent de fois : la comparaison de $i$ avec $n$ se produit à $n + 1$ reprises, tandis que l'incrémentation de $i$ se passe uniquement

1. Si vous avez des connaissances en architecture des ordinateurs, vous savez que le temps mis pour accéder à une variable ou à l'élément d'un tableau n'est pas nécessairement fixe et qu'il peut dépendre du fait que la variable ou l'élément du tableau se trouve dans le cache, en mémoire principale ou sur le disque (système à mémoire virtuelle). Certains modèles sophistiqués tiennent compte de ces aspects, mais il est généralement suffisant de supposer que toutes les variables et les tableaux se trouvent en mémoire principale et que les accès se font donc tous avec la même durée.

Copyright © 2013 Dunod.

$n$ fois. Séparons le temps de l'étape 2 en $t_2'$ pour le test et en $t_2''$ pour l'incrémentation. De même, séparons le temps de l'étape 2A en $t_{2A}'$ pour le test $A[i] = x$ et en $t_{2A}''$ pour l'affectation de $i$ à *réponse*. Le temps d'exécution de RECHERCHE-LINÉAIRE se situe donc entre

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot 0 + t_3$$

et

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3$$

Nous récrivons ces deux bornes, en regroupant les termes ayant pour facteur $n$ et les autres. Le temps d'exécution se situe entre la **borne inférieure**

$$(t_2' + t_2'' + t_{2A}') \cdot n + (t_1 + t_2' + t_3)$$

et la **borne supérieure**

$$(t_2' + t_2'' + t_{2A}' + t_{2A}'') \cdot n + (t_1 + t_2' + t_3)$$

Vous remarquerez que ces deux bornes sont de la forme $c \cdot n + d$, où $c$ et $d$ sont des constantes qui ne dépendent pas de $n$. Autrement dit, les deux bornes sont des *fonctions linéaires* de $n$. Le temps d'exécution de la procédure RECHERCHE-LINÉAIRE est donc borné par une borne inférieure qui est une fonction linéaire de $n$ et par une borne supérieure qui est une fonction linéaire de $n$.

Il existe une notation particulière pour indiquer qu'un temps d'exécution présente des bornes inférieure et supérieure qui sont des fonctions linéaires (éventuellement différentes) de $n$. Nous écrivons que le temps d'exécution est $\Theta(n)$. Il s'agit de la lettre grecque thêta et nous disons « thêta de $n$ » ou simplement « thêta $n$ ». Comme promis au Chapitre 1, cette notation omet le terme d'ordre inférieur $(t_1 + t_2' + t_3)$ et les coefficients de $n(t_2' + t_2'' + t_{2A}'')$ pour la borne inférieure et $t_2' + t_2'' + t_{2A}' + t_{2A}''$ pour la borne supérieure). Même si en caractérisant le temps d'exécution à $\Theta(n)$ nous perdons en précision, nous avons l'avantage de mettre en avant son ordre de grandeur et de supprimer les détails fastidieux.

La notation $\Theta$ s'applique aux fonctions en général et pas uniquement à celles qui décrivent le temps d'exécution des algorithmes, ainsi qu'aux fonctions autres que linéaires. L'idée est la suivante : si nous avons deux fonctions, $f(n)$ et $g(n)$, nous disons que $f(n)$ est $\Theta(g(n))$ si $f(n)$ est bornée par un facteur constant de $g(n)$ pour $n$ suffisamment grand. Nous pouvons donc dire que le temps d'exécution de RECHERCHE-LINÉAIRE est borné par un facteur constant de $n$ dès que $n$ est suffisamment grand.

Il existe une définition technique de la notation $\Theta$ plutôt intimidante, mais, heureusement, nous devons rarement y avoir recours pour utiliser cette notation. Nous nous focalisons simplement sur le terme dominant, en omettant les termes d'ordre inférieur et les facteurs constants. Par exemple, la fonction $n^2 / 4 + 100n + 50$ est $\Theta(n^2)$ ; dans ce cas, nous retirons les

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

termes d'ordre inférieur $100n$ et 50, ainsi que le facteur constant $1 / 4$. Bien que les termes d'ordre inférieur auront plus d'importance que $n^2 / 4$ pour les petites valeurs de $n$, dès que celui-ci dépasse 400, le terme $n^2 / 4$ devient supérieur à $100n + 50$. Lorsque $n = 1\,000$, le terme dominant $n^2 / 4$ est égal à 250 000, tandis que les termes d'ordre inférieur $100n + 50$ arrivent uniquement à 100 050 ; pour $n = 2\,000$, la comparaison se fait entre 1 000 000 et 200 050. Dans le monde des algorithmes, nous détournons légèrement la notation et écrivons $f(n) = \Theta(g(n))$ afin que nous puissions écrire $n^2 / 4 + 100n + 50 = \Theta(n^2)$.

Étudions à présent le temps d'exécution de MEILLEURE-RECHERCHE-LINÉAIRE de la page 15. La tâche est un peu plus ardue que pour RECHERCHE-LINÉAIRE car nous ne pouvons pas connaître à l'avance le nombre d'itérations de la boucle. Si $A[1]$ est égal à $x$, il y aura une seule itération. Si $x$ est absent du tableau, la boucle sera exécutée à $n$ reprises, ce qui correspond au maximum possible. Chaque itération se fait en un temps constant et nous pouvons dire que, *dans le cas le plus défavorable*, MEILLEURE-RECHERCHE-LINÉAIRE prend un temps $\Theta(n)$ pour effectuer la recherche dans un tableau de $n$ éléments. Pourquoi « dans le cas le plus défavorable » ? Puisque nous voulons que les algorithmes aient des temps d'exécution faibles, le cas le plus défavorable se produit lorsqu'un algorithme prend le temps maximal quelle que soit l'entrée.

Dans le cas le plus favorable, c'est-à-dire lorsque $A[1]$ est égal à $x$, MEILLEURE-RECHERCHE-LINÉAIRE prend un temps constant : la procédure fixe $i$ à 1, vérifie si $i \leq n$, le test $A[i] = x$ devient vrai et la valeur de $i$, dans ce cas 1, est retournée. Ce temps ne dépend donc pas de $n$. Nous écrivons que le *temps d'exécution dans le cas le plus favorable* de MEILLEURE-RECHERCHE-LINÉAIRE est $\Theta(1)$ car, dans le cas optimal, son temps d'exécution reste dans un facteur constant de 1. Autrement dit, le temps d'exécution dans le cas le plus favorable est une constante qui ne dépend pas de $n$.

Vous l'avez compris, nous ne pouvons donc pas employer la notation $\Theta$ de façon générale pour couvrir tous les cas de temps d'exécution de la procédure MEILLEURE-RECHERCHE-LINÉAIRE. NOUS NE SOMMES PAS EN MESURE D'ÉTABLIR QUE LE TEMPS D'EXÉCUTION EST TOUJOURS $\Theta(n)$ car, dans le cas le plus favorable, il est $\Theta(1)$. Il est également impossible de dire qu'il est toujours $\Theta(1)$ car, dans le cas le plus défavorable, il est $\Theta(n)$. En revanche, nous pouvons indiquer qu'une fonction linéaire de $n$ représente dans tous les cas une *borne supérieure* et nous avons pour cela la notation suivante : $O(n)$. À l'oral, nous la prononçons « grand o de $n$ » ou, plus simplement « o de $n$ ». Une fonction $f(n)$ est $O(g(n))$ si, avec $n$ suffisamment grand, la borne supérieure de $f(n)$ est une constante multipliée par $g(n)$. De nouveau, nous procédons à un petit abus de notation et écrivons $f(n) = O(g(n))$. Dans le cas de MEILLEURE-RECHERCHE-LINÉAIRE, nous

Copyright © 2013 Dunod.

pouvons généraliser son temps d'exécution et dire qu'il est $O(n)$ ; bien que le temps d'exécution puisse être meilleur qu'une fonction linéaire de $n$, il ne sera jamais pire.

La notation $O$ est employée pour indiquer qu'un temps d'exécution n'est jamais *pire* qu'une constante multipliée par une fonction de $n$, mais comment pouvons-nous préciser qu'il n'est jamais *meilleur* qu'une constante multipliée par une fonction de $n$ ? Il s'agit d'une borne inférieure et nous utilisons la notation $\Omega$, reflet de la notation $O$ : une fonction $f(n)$ est $\Omega(g(n))$ si, avec $n$ suffisamment grand, la borne inférieure de $f(n)$ est une constante multipliée par $g(n)$. Nous disons que « $f(n)$ est grand oméga de $g(n)$ » ou, plus simplement, que « $f(n)$ est oméga de $g(n)$ », et nous pouvons écrire $f(n) = \Omega(g(n))$. Puisque la notation $O$ donne une borne supérieure, que la notation $\Omega$ donne une borne inférieure et que la notation $\Theta$ donne les deux, nous pouvons conclure qu'une fonction $f(n)$ est $\Theta(g(n))$ si et seulement si $f(n)$ est à la fois $O(g(n))$ et $\Omega(g(n))$.

Nous pouvons faire une déclaration générale sur la borne inférieure du temps d'exécution de MEILLEURE-RECHERCHE-LINÉAIRE : dans tous les cas il est $\Omega(1)$. Cette déclaration est un tantinet légère, car nous nous attendons évidemment à ce que n'importe quel algorithme appliqué à n'importe quelle entrée ait un temps d'exécution au moins constant. Nous emploierons peu la notation $\Omega$, mais elle se révélera pratique à l'occasion.

Pour décrire les notations $\Theta$, $O$ et $\Omega$, le terme générique est **notation asymptotique**. En effet, ces notations représentent la croissance d'une fonction lorsque son argument tend vers l'infini de façon asymptotique. Grâce à ces notations asymptotiques, nous pouvons omettre les termes d'ordre inférieur et les facteurs constants. Cela nous permet d'ignorer les détails fastidieux et de nous focaliser sur l'important : la croissance de la fonction avec $n$.

Revenons à présent à la procédure RECHERCHE-LINÉAIRE-SENTINELLE de la page 17. Comme pour MEILLEURE-RECHERCHE-LINÉAIRE, chaque itération de sa boucle se fait en un temps constant et le nombre d'itérations varie entre 1 et $n$. La différence principale entre RECHERCHE-LINÉAIRE-SENTINELLE et MEILLEURE-RECHERCHE-LINÉAIRE est que le temps d'une itération de la première est inférieur à celui de la seconde. Dans le cas le plus défavorable, ces deux temps sont linéaires, mais le facteur constant de RECHERCHE-LINÉAIRE-SENTINELLE est meilleur. Même si nous nous attendons à ce que la procédure RECHERCHE-LINÉAIRE-SENTINELLE soit dans la pratique plus rapide, elle ne le sera que d'un facteur constant. Lorsque nous exprimons les temps d'exécution de ces procédures en utilisant une notation asymptotique, ils sont équivalents : $\Theta(n)$ dans le cas le plus défavorable, $\Theta(1)$ dans le cas le plus favorable et $O(n)$ dans tous les cas.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

## Invariants de boucle

Dans nos trois variantes de la recherche linéaire, il est facile de voir qu'elles produisent toutes une réponse exacte. Parfois, cela sera plus difficile. Il existe de nombreuses techniques pour le savoir, plus que je ne peux en décrire dans cet ouvrage.

Pour prouver l'exactitude, une méthode répandue se fonde sur un *invariant de boucle* : il s'agit d'une affirmation dont nous montrons la véracité à chaque début d'une itération de boucle. Pour qu'un invariant de boucle nous aide à vérifier l'exactitude, nous devons montrer les trois points suivants :

1. **Initialisation :** l'invariant est vrai avant la première itération de la boucle.
2. **Conservation :** si l'invariant est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
3. **Terminaison :** la boucle se termine et, à ce moment-là, l'invariant de boucle, ainsi que la raison de la fin de la boucle, nous fournissent une propriété utile.

Voici par exemple l'invariant de boucle pour Meilleure-Recherche-Linéaire :

> Au début de chaque itération de l'étape 1, si $x$ est présent dans le tableau $A$, alors il est présent dans le **sous-tableau** (une portion contiguë d'un tableau) qui va de $A[i]$ à $A[n]$.

Nous n'avons pas besoin de cet invariant de boucle pour montrer que si la procédure retourne une valeur autre que Non-Trouvé, alors l'indice retourné est correct : la seule raison pour que la procédure retourne un indice $i$ à l'étape 1A est que $x$ soit égal à $A[i]$. À la place, nous utiliserons cet invariant de boucle pour montrer que si la procédure retourne Non-Trouvé à l'étape 2, alors $x$ est absent du tableau :

1. **Initialisation :** initialement $i = 1$ et le sous-tableau dans l'invariant de boucle va de $A[1]$ à $A[n]$, ce qui correspond à l'intégralité du tableau.
2. **Conservation :** supposons qu'au début d'une itération pour une valeur de $i$, si $x$ est présent dans le tableau $A$, alors il l'est également dans le sous-tableau qui va de $A[i]$ à $A[n]$. Si l'itération se déroule sans que la procédure ne retourne, nous savons que $A[i] \neq x$, et nous pouvons donc affirmer que si $x$ est présent dans le tableau $A$, alors il est présent dans le sous-tableau qui va de $A[i + 1]$ à $A[n]$. Puisque $i$ est incrémenté avant l'itération suivante, l'invariant de boucle est conservé avant la prochaine itération.
3. **Terminaison :** cette boucle doit se terminer, soit parce que la procédure retourne à l'étape 1A, soit parce que $i > n$. Nous avons déjà traité le cas où la boucle se termine en raison d'un retour lors de l'étape 1A.

   Dans le cas où la boucle se termine parce que $i > n$, nous nous fondons sur la contraposée de l'invariant de boucle. La *contraposée* de la

Copyright © 2013 Dunod.

proposition « A implique B » est « non-B implique non-A ». La contraposée d'une proposition est vraie si et seulement si la proposition est vraie. La contraposée de l'invariant de boucle est « si $x$ n'est pas présent dans le sous-tableau de $A[i]$ à $A[n]$, alors il n'est pas présent dans le tableau $A$ ».

À présent, lorsque $i > n$, le sous-tableau de $A[i]$ à $A[n]$ est vide et ne peut donc pas contenir $x$. En raison de la contraposée de l'invariant de boucle, $x$ est donc absent du tableau $A$ et il est approprié de retourner NON-TROUVÉ à l'étape 2.

Cela fait un raisonnement plutôt important pour ce qui n'est qu'une simple boucle ! Devons-nous le tenir chaque fois que nous écrivons une boucle ? Ce n'est pas mon cas, mais certaines personnes insistent sur un tel raisonnement rigoureux pour n'importe quelle boucle. Lorsque je programme, je constate que j'ai souvent un invariant de boucle quelque part dans ma tête. Il est probablement tellement enfoui que je ne le perçois pas, mais je pourrais le donner si jamais cela était nécessaire. Bien que la plupart d'entre nous seraient d'accord pour dire qu'un invariant de boucle est un peu exagéré pour comprendre la boucle simple de MEILLEURE-RECHERCHE-LINÉAIRE, les invariants de boucle se révéleront plutôt pratiques pour comprendre pourquoi des boucles plus complexes font correctement leur travail.

## RÉCURSION

Avec la *récursion*, un problème est résolu en résolvant de plus petites instances du même problème. Voici mon exemple préféré de récursion : calculer $n!$ (factorielle $n$), qui est définie pour les valeurs positives de $n$ par :

$n! = 1$ si $n = 0$

et

$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$ si $n \geq 1$.

Par exemple, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Nous observons que

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

et donc que $n! = n \cdot (n-1)!$ pour $n \geq 1$.

Nous avons défini $n!$ en fonction d'un problème de taille plus réduite, c'est-à-dire $(n-1)!$. Nous pouvons écrire une procédure récursive pour calculer $n!$ :

*Procédure* FACTORIELLE($n$)
*Entrée :* Un entier $n \geq 0$.
*Sortie :* La valeur de $n!$.

1. Si $n = 0$, alors retourner 1 comme sortie.
2. Sinon retourner $n$ fois la valeur obtenue en appelant de façon récursive FACTORIELLE($n-1$).

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Ma façon de rédiger l'étape 2 est plutôt lourde. J'aurais pu simplement écrire à la place « Sinon retourner $n \cdot$ FACTORIELLE$(n-1)$ », en utilisant la valeur de retour de l'appel récursif dans une expression arithmétique plus longue.

Pour que la récursion opère, deux propriétés doivent être satisfaites. Premièrement, il faut au moins un **cas de base**, dans lequel nous calculons la solution directement sans passer par la récursion. Deuxièmement, chaque appel récursif de la procédure doit se faire sur une *instance plus petite du même problème*, pour finir par atteindre un cas de base. Pour la procédure FACTORIELLE, le cas de base correspond à $n = 0$, et chaque appel récursif se fait sur une instance dans laquelle la valeur de $n$ est diminuée de 1. Tant que la valeur initiale de $n$ est positive, les appels récursifs finissent par arriver au cas de base.

Affirmer qu'un algorithme récursif fonctionne peut sembler au premier abord extrêmement simple. Le point clé est de croire que chaque appel récursif produit le résultat correct. Tant que nous sommes prêts à croire que les appels récursifs effectuent correctement leur travail, il est souvent facile de soutenir l'exactitude de l'algorithme. Voici comment nous pouvons affirmer que la procédure FACTORIELLE retourne la réponse exacte. Lorsque $n = 0$, il est évident que la valeur retournée, 1, est égale à $n!$. Nous supposons que, lorsque $n \geq 1$, l'appel récursif à FACTORIELLE$(n-1)$ travaille correctement et retourne la valeur de $(n-1)!$. La procédure multiplie ensuite cette valeur par $n$, et calcule donc la valeur de $n!$, qu'elle finit par retourner.

Dans l'exemple suivant, les appels récursifs ne se font pas sur de plus petites instances du même problème, alors que les mathématiques sont justes. Il est évident que si $n \geq 0$, alors $n! = (n+1)!/(n+1)$. Cependant, la procédure récursive suivante, qui exploite cette formule, ne parvient pas à donner une réponse lorsque $n \geq 1$ :

*Procédure* FACTORIELLE-ERRONÉE$(n)$
*Entrées et sortie :* Identiques à celle de FACTORIELLE.
1. Si $n = 0$, alors retourner 1 comme sortie.
2. Sinon retourner FACTORIELLE-ERRONÉE$(n+1) / (n+1)$.

Un appel à FACTORIELLE-ERRONÉE(1) génère un appel récursif à FACTORIELLE-ERRONÉE(2), qui génère un appel récursif à FACTORIELLE-ERRONÉE(3), etc. Nous n'arrivons donc jamais au cas de base pour lequel $n$ est égal à 0. En implémentant cette procédure à l'aide d'un langage de programmation réel et en exécutant sur un ordinateur, nous obtenons rapidement un message du type « erreur de dépassement de pile ».

Copyright © 2013 Dunod.

Nous pouvons souvent récrire sous forme récursive un algorithme qui met en œuvre une boucle. Voici une recherche linéaire, sans la sentinelle, récrite de façon récursive :

*Procédure* RECHERCHE-LINÉAIRE-RÉCURSIVE$(A,n,i,x)$

*Entrées :* Identiques à celles de RECHERCHE-LINÉAIRE, mais en ajoutant le paramètre $i$.

*Sortie :* L'indice d'un élément égal à $x$ présent dans le sous-tableau qui va de $A[i]$ à $A[n]$, ou NON-TROUVÉ si $x$ est absent de ce sous-tableau.

1. Si $i > n$, alors retourner NON-TROUVÉ.
2. Sinon ($i \leq n$), si $A[i] = x$, alors retourner $i$.
3. Sinon ($i \leq n$ et $A[i] \neq x$), retourner RECHERCHE-LINÉAIRE-RÉCURSIVE$(A,n,i+1,x)$.

Dans cet exemple, le sous-problème est de rechercher $x$ dans le sous-tableau qui va de $A[i]$ à $A[n]$. Le cas de base est rencontré à l'étape 1, lorsque ce sous-tableau est vide, c'est-à-dire lorsque $i > n$. Puisque la valeur de $i$ augmente lors de chaque appel récursif effectué à l'étape 3, si aucun appel récursif n'a retourné une valeur de $i$ à l'étape 2, alors $i$ finit par devenir supérieur à $n$ et nous arrivons au cas de base.

## LECTURES COMPLÉMENTAIRES

Les Chapitres 2 et 3 du CLRS [CLRS09] couvrent une grande partie du contenu de celui-ci. Un des premiers ouvrages sur l'algorithmique, écrit par Aho, Hopcroft et Ullman [AHU74], a largement contribué à l'utilisation de la notation asymptotique dans l'analyse des algorithmes. De nombreux travaux ont été menés pour prouver la justesse des programmes. Si ce domaine vous intéresse, consultez ceux de Gries [Gri81] et de Mitchell [Mit96].

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# 3 ALGORITHMES DE TRI ET DE RECHERCHE

Le Chapitre 2 a présenté trois variantes de la recherche linéaire appliquée à un tableau. Pouvons-nous faire mieux ? La réponse est : cela dépend. Si nous ne connaissons rien sur l'ordre des éléments du tableau, alors non nous ne pouvons pas faire mieux. Dans le cas le plus défavorable, nous devons parcourir l'ensemble des $n$ éléments car, si nous ne trouvons pas la valeur recherchée dans les $n - 1$ premiers, elle peut être dans le dernier, le $n$-ième. Par conséquent, si nous n'avons aucune information sur l'ordre des éléments dans le tableau, nous ne pouvons pas obtenir, dans le cas le plus défavorable, un meilleur temps d'exécution que $\Theta(n)$.

En revanche, supposons que le tableau soit trié par ordre croissant : chaque élément est inférieur ou égal à son successeur dans le tableau, conformément à la définition donnée à « inférieur à ». Dans ce chapitre, nous verrons que si un tableau est trié, alors nous pouvons employer une technique simple, appelée recherche dichotomique, pour effectuer une recherche dans un tableau de $n$ éléments en un temps de seulement $O(\lg n)$. Nous l'avons vu au Chapitre 1, la valeur de $\lg n$ croît très lentement en fonction de $n$. Dans le cas le plus défavorable, une recherche dichotomique est donc plus efficace qu'une recherche linéaire[1].

Pour un élément, qu'est-ce que signifie être inférieur à un autre ? Lorsque les éléments sont des nombres, le sens est évident. Lorsque les éléments sont des chaînes de caractères, nous pouvons imaginer un **ordre lexicographique** : un élément est inférieur à un autre s'il vient avant celui-ci dans le dictionnaire. Lorsque les éléments sont des données d'autre forme, nous devons donner la définition de « inférieur à ». Tant qu'elle est claire, nous pouvons déterminer si un tableau est trié.

1. Si vous n'êtes pas informaticien et avez sauté la section « Algorithmes pour les non-informaticiens » du Chapitre 1, vous devriez lire les explications qui concernent les logarithmes à la page 7.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Dans l'exemple des ouvrages sur l'étagère du Chapitre 2, nous avons dit que nous pouvons trier les livres par ordre alphabétique d'auteur ou de titre ou, dans une bibliothèque, en fonction d'une référence. Dans ce chapitre, les livres sont triés s'ils apparaissent sur l'étagère par ordre alphabétique d'auteur, en allant de gauche à droite. Cependant, plusieurs ouvrages du même auteur peuvent se trouver sur l'étagère, par exemple de William Shakespeare. Imaginons que nous voulions rechercher non pas n'importe quel ouvrage de Shakespeare mais un livre précis. Dans ce cas, nous disons que si deux livres ont le même auteur, alors celui dont le titre se trouve alphabétiquement en premier doit être placé à gauche. Nous pouvons également dire que seul le nom de l'auteur nous intéresse et, lors de la recherche, n'importe quel ouvrage de Shakespeare conviendra donc. Les informations que nous utilisons pour les correspondances sont appelées **clés**. Dans notre exemple de bibliothèque personnelle, la clé est uniquement le nom de l'auteur, plutôt qu'une combinaison fondée d'abord sur le nom de l'auteur puis sur le titre en cas de deux ouvrages du même auteur.

Comment pouvons-nous obtenir un tableau trié ? Dans ce chapitre, nous allons étudier quatre algorithmes – tri par sélection, tri par insertion, tri par fusion et tri rapide – qui permettent de trier un tableau, en appliquant chacun d'eux à notre exemple de bibliothèque. Chaque algorithme a ses avantages et ses inconvénients, ce qui nous amènera, à la fin de ce chapitre, à les analyser et à les comparer. Tous les algorithmes de tri étudiés dans ce chapitre ont dans le cas le plus défavorable un temps d'exécution $\Theta(n^2)$ ou $\Theta(n \lg n)$. Si vous souhaitez uniquement effectuer quelques recherches, il est donc préférable de réaliser une recherche linéaire. En revanche, pour des recherches multiples, la meilleure solution sera de commencer par trier le tableau, puis de mettre en place une recherche dichotomique.

Le tri est en soi un problème important et pas seulement en tant qu'étape préalable à une recherche dichotomique. Réfléchissez à toutes les informations qui doivent être triées, comme les noms dans les annuaires téléphoniques, les chèques (par numéro et/ou par date) dans les relevés bancaires et les résultats d'un moteur de recherche web (par degré de pertinence). Par ailleurs, le tri constitue souvent une étape au sein d'un autre algorithme. Ainsi, dans un logiciel d'infographie, les objets sont souvent superposés. Le code d'affichage des objets à l'écran devra probablement les trier conformément à une relation « au-dessus de » afin de les dessiner du bas vers le haut.

Avant de poursuivre, nous devons nous intéresser à ce que nous trions. Outre la clé (appelée **clé de tri** dans le cas d'une opération de tri), les éléments triés comprennent généralement des **données satellites**. Bien que les données satellites puissent provenir d'un satellite, ce n'est généralement pas le cas. Il s'agit des informations qui sont associées à la clé de tri et qui doivent être transférées avec elle lorsque des éléments sont déplacés. Dans

Copyright © 2013 Dunod.

notre exemple d'étagère de livres, la clé de tri correspond au nom de l'auteur et les données satellites à l'ouvrage lui-même.

À mes étudiants, je présente les données satellites d'une façon qui me permet d'être certain qu'ils comprennent. Je prends une feuille de calcul avec leurs notes, les lignes étant triées par ordre alphabétique sur leur nom. Pour déterminer la note finale, je trie des lignes. La clé de tri correspond à la colonne qui contient la note obtenue dans une matière, les autres colonnes, y compris le nom de l'étudiant, constituant les données satellites. Le tri se fait par ordre décroissant de la note. Les premières lignes comprennent donc les meilleures notes, les dernières les plus mauvaises. Supposons que seules les informations de la colonne des notes soient réordonnées, sans déplacer l'intégralité du contenu de chaque ligne. Les noms des étudiants restent alors par ordre alphabétique, quelle que soit leur note. Ceux dont le nom apparaît au début de l'alphabet seront donc des étudiants heureux, tandis que ceux qui arrivent en fin d'alphabet le seront beaucoup moins.

Prenons quelques autres exemples de clés de tri et de données satellites. Dans un annuaire, la clé de tri peut être le nom et les données satellites, l'adresse et le numéro de téléphone. Sur un relevé bancaire, la clé de tri peut être le numéro de chèque et les données satellites, le montant du chèque et la date de valeur. Dans un moteur de recherche, la clé de tri peut être le niveau de pertinence de la requête et les données satellites, l'URL de la page web, ainsi que les autres informations affichées par le moteur.

Dans ce chapitre, nous manipulerons les tableaux comme si chaque élément contenait uniquement une clé de tri. Si vous souhaitez implémenter les algorithmes décrits, vous devrez faire attention à déplacer les données satellites associées à chaque élément, ou au moins un pointeur sur ces données, en même temps que la clé de tri.

Pour que l'analogie avec l'étagère de livres tienne, nous devons supposer qu'elle et les ouvrages qui se trouvent dessus possèdent deux caractéristiques supplémentaires ; j'admets leur manque de réalisme. Tout d'abord, tous les ouvrages ont la même taille, car, dans un tableau informatique, toutes les entrées ont la même taille. Ensuite, nous pouvons numéroter l'emplacement de chaque ouvrage sur l'étagère de 1 à $n$, et chaque emplacement est appelé *alvéole*. L'alvéole 1 se situe à l'extrême gauche, tandis que l'alvéole $n$ se trouve à l'extrême droite. Vous l'aurez deviné, chaque alvéole de l'étagère correspond à une entrée du tableau.

Je voudrais également revenir sur le terme « trier ». Dans le langage courant, il peut avoir un sens différent de celui qui lui est donné en informatique. Mon dictionnaire en ligne définit cette action comme « séparer pour répartir et regrouper selon certains critères ». Cela correspond par exemple à trier des vêtements, avec les pantalons sur une étagère, les chemises sur une autre, etc. Dans le monde des algorithmes informatiques,

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

trier signifie placer dans un ordre défini, tandis que « répartir et regrouper selon certains critères » se dit « classifier ».

## Recherche dichotomique

Avant d'examiner les algorithmes de tri, étudions la recherche dichotomique (ou recherche binaire), qui exige un tableau trié. Son avantage est un temps d'exécution $O(\lg n)$ pour une recherche dans un tableau de $n$ éléments.

Dans notre exemple d'étagère de livres, nous supposons que les ouvrages sont déjà triés de gauche à droite par nom d'auteur. Nous utilisons celui-ci comme clé et recherchons n'importe quel ouvrage écrit par Jonathan Swift. Puisque le nom de famille de l'auteur commence par la lettre « S », qui se trouve en 19e position dans l'alphabet, nous pourrions aller aux environs des trois quarts de l'étagère (19 / 26 est proche de 3 / 4) et examiner l'alvéole correspondante. Toutefois, si notre collection comprend tous les livres de Shakespeare, nous aurons plusieurs livres dont le nom de famille de l'auteur vient avant Swift et ils repousseront les ouvrages de Swift vers la droite, plus loin qu'attendu.

Oublions cette méthode et appliquons de la manière suivante la recherche dichotomique d'un ouvrage de Jonathan Swift. Nous nous positionnons au milieu de l'étagère et examinons le nom de l'auteur de l'ouvrage qui s'y trouve. Supposons qu'il ait été écrit par Jack London. Nous savons non seulement que ce livre ne nous intéresse pas, mais également que, puisque les ouvrages sont triés par ordre alphabétique sur le nom de l'auteur, tous ceux qui se trouvent à sa gauche ne nous intéressent pas plus. En examinant un seul livre, nous avons éliminé la moitié des ouvrages de l'étagère ! Un livre de Swift se trouvera donc en partie droite de l'étagère. Nous allons ensuite à l'alvéole qui se trouve au milieu de la moitié droite et examinons l'ouvrage correspondant. Supposons qu'il ait été écrit par Léon Tolstoï. À nouveau, ce livre ne nous intéresse pas, mais nous pouvons éliminer tous ceux qui se trouvent à sa droite. À ce stade, si notre étagère comprend un ouvrage de Swift, il doit se trouver dans le quart situé à droite du livre de London et à gauche de celui de Tolstoï. Notre analyse se poursuit au milieu de cette portion identifiée. Si l'ouvrage correspondant est de Swift, la recherche est terminée. Sinon nous pouvons à nouveau éliminer la moitié des ouvrages restants. Nous finissons par trouver un livre de Swift ou arrivons au point où il ne reste plus aucune alvéole. Nous en concluons alors que l'étagère ne comprend aucun ouvrage écrit par Jonathan Swift.

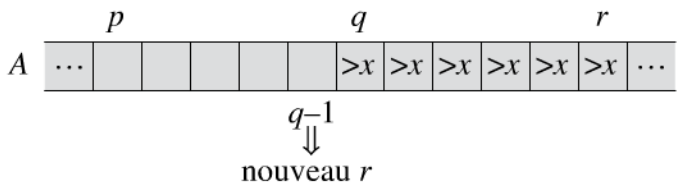
Dans un ordinateur, la recherche dichotomique se fait sur un tableau. À tout moment, nous examinons uniquement un sous-tableau, c'est-à-dire

Copyright © 2013 Dunod.

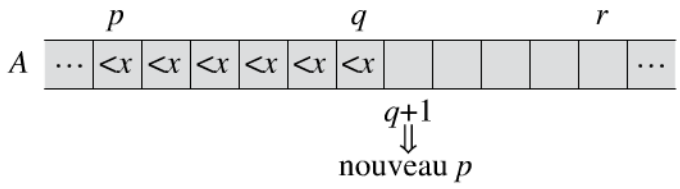
la partie du tableau qui se trouve entre deux indices, ceux-ci inclus ; nommons-les $p$ et $r$. Initialement $p = 1$ et $r = n$. Le sous-tableau correspond donc au tableau entier. Nous répétons la division en deux de la taille du sous-tableau examiné, jusqu'à ce que nous trouvions la valeur recherchée ou que le sous-tableau soit vide (lorsque $p$ devient supérieur à $r$). Le temps d'exécution $O(\lg n)$ est lié aux multiples séparations en deux.

Voici une explication plus détaillée du fonctionnement de la recherche dichotomique. Supposons que nous recherchions la valeur $x$ dans le tableau $A$. À chaque étape, nous examinons uniquement le sous-tableau qui commence à $A[p]$ et se termine à $A[r]$. Puisque nous allons souvent manipuler des sous-tableaux, utilisons la notation $A[p..r]$. Nous déterminons le point central $q$ du sous-tableau examiné en calculant la moyenne de $p$ et de $r$, puis en ignorant la partie fractionnaire, si elle existe : $q = \lfloor (p + r) / 2 \rfloor$. (Nous utilisons la notation $\lfloor\ \rfloor$ pour indiquer l'opération « partie entière ». Si vous l'implémentez dans un langage comme Java, C ou C++, vous pouvez simplement utiliser une division entière pour retirer la partie fractionnaire.) Nous vérifions si $A[q]$ est égal à $x$. Dans l'affirmative, nous avons terminé, car nous pouvons simplement retourner $q$ comme indice de l'emplacement où se trouve $x$ dans le tableau $A$.

Si, à la place, $A[q] \neq x$, alors nous exploitons l'hypothèse que le tableau $A$ est déjà trié. Puisque $A[q] \neq x$, nous avons deux possibilités : $A[q] > x$ ou $A[q] < x$. Nous commençons par traiter le cas où $A[q] > x$. Puisque le tableau est trié, nous savons non seulement que $A[q]$ est supérieur à $x$, mais également que, en imaginant le tableau disposé de gauche à droite, chaque élément à droite de $A[q]$ est supérieur à $x$. Par conséquent, nous pouvons ne pas prendre en considération tous les éléments qui se trouvent à $A[q]$ et à sa droite. Pour l'étape suivante, nous ne touchons pas à $p$, mais $r$ prend la valeur de $q - 1$ :




Si nous avons $A[q] < x$, nous savons que chaque élément du tableau à l'emplacement $A[q]$ et à sa gauche est inférieur à $x$. Nous pouvons donc ne pas prendre en compte ces éléments. L'étape suivante débutera ainsi avec un indice $r$ non modifié, mais un indice $p$ fixé à $q + 1$ :




Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Voici la procédure de la recherche dichotomique :

*Procédure* RECHERCHE-DICHOTOMIQUE($A$,$n$,$x$)
*Entrées et sortie :* Identiques à RECHERCHE-LINÉAIRE.
1. Fixer $p$ à 1 et $r$ à $n$.
2. Tant que $p \leq r$, faire la chose suivante :
   A. Fixer $q$ à $\lfloor (p + r) / 2 \rfloor$.
   B. Si $A[q] = x$, alors retourner $q$.
   C. Sinon ($A[q] \neq x$) si $A[q] > x$, alors fixer $r$ à $q - 1$.
   D. Sinon ($A[q] < x$) fixer $p$ à $q + 1$.
3. Retourner NON-TROUVÉ.

La boucle de l'étape 2 ne se termine pas nécessairement parce que $p$ devient supérieur à $r$. Cela peut également se produire à l'étape 2B car, si $A[q]$ est égal à $x$, la procédure retourne $q$ en tant qu'indice de l'emplacement de $x$ dans le tableau $A$.

Pour montrer que le fonctionnement de la procédure est correct, nous devons simplement montrer que $x$ est absent du tableau si RECHERCHE-DICHOTOMIQUE retourne NON-TROUVÉ à l'étape 3. Pour cela, nous utilisons l'invariant de boucle suivant :

> Au début de chaque itération de la boucle de l'étape 2, si $x$ est présent dans le tableau $A$, alors il se trouve quelque part dans le sous-tableau $A[p..r]$.

Voici l'argumentation :

- **Initialisation** : l'étape 1 initialise les indices $p$ à 1 et $r$ à $n$. L'invariant de boucle est donc vrai lorsque la procédure entre pour la première fois dans la boucle.
- **Conservation** : nous avons expliqué précédemment que les étapes 2C et 2D ajustent correctement $p$ ou $r$.
- **Terminaison** : si $x$ n'est pas dans le tableau, alors la procédure finit par arriver au point où $p$ et $r$ sont égaux. Lorsque cela se produit, l'étape 2A fixe $q$ à la même valeur que $p$ et $r$. Si l'étape 2C fixe $r$ à $q - 1$, alors, au début de l'itération suivante, $r$ est égal à $p - 1$ et $p$ est donc supérieur à $r$ . Si l'étape 2D fixe $p$ à $q + 1$, alors, au début de l'itération suivante, $p$ est égal à $r + 1$, et à nouveau $p$ est supérieur à $r$. Dans les deux cas, le test de la boucle de l'étape 2 devient faux et la boucle se termine. Puisque $p > r$, le sous-tableau $A[p..r]$ est vide et la valeur $x$ est nécessairement absente. En prenant la contraposée de l'invariant de boucle (voir page 22), nous savons que si $x$ est absent du sous-tableau $A[p..r]$, alors il n'est pas dans le tableau $A$. Par conséquent, le retour de la valeur NON-TROUVÉ à l'étape 3 correspond à un fonctionnement correct de la procédure.

Copyright © 2013 Dunod.

Nous pouvons également écrire la recherche dichotomique sous forme d'une procédure récursive :

*Procédure* RECHERCHE-DICHOTOMIQUE-RÉCURSIVE$(A, p, r, x)$

*Entrées et sortie :* Les entrées $A$ et $x$ sont identiques à celles de RECHERCHE-LINÉAIRE, ainsi que la sortie. Les entrées $p$ et $r$ balisent le sous-tableau $A[p..r]$ examiné.

1. Si $p > r$, alors retourner NON-TROUVÉ.
2. Sinon ($\leq pr$), faire la chose suivante :
   A. Fixer $q$ à $\lfloor (p + r) / 2 \rfloor$.
   B. Si $A[q] = x$, alors retourner $q$.
   C. Sinon ($A[q] \neq x$), si $A[q] > x$ alors retourner RECHERCHE-DICHOTOMIQUE-RÉCURSIVE$(A, p, q - 1, x)$.
   D. Sinon ($A[q] < x$) retourner RECHERCHE-DICHOTOMIQUE-RÉCURSIVE$(A, q + 1, r, x)$.

L'appel initial est RECHERCHE-DICHOTOMIQUE-RÉCURSIVE$(A, 1, n, x)$.

Expliquons à présent pourquoi la recherche dichotomique a un temps d'exécution $O(\lg n)$ sur un tableau de $n$ éléments. Pour cela, il faut remarquer que la taille $r - p + 1$ du sous-tableau examiné est approximativement divisée par deux à chaque itération de la boucle (ou à chaque appel récursif dans la version correspondante, mais limitons-nous à la version itérative de RECHERCHE-DICHOTOMIQUE). En essayant tous les cas, nous constatons que si l'itération débute avec un sous-tableau de $s$ éléments, l'itération suivante comprend $\lfloor s / 2 \rfloor$ ou $s / 2 - 1$ éléments, selon que $s$ est pair ou impair et que $A[q]$ est supérieur ou inférieur à $x$. Nous avons déjà vu que lorsque la taille du tableau arrive à 1, la procédure s'arrête à l'itération suivante. Nous pouvons donc nous demander combien d'itérations de la boucle il faut pour découper en deux un sous-tableau depuis sa taille d'origine $n$ à la taille 1. Le résultat sera identique au nombre de fois que, en partant d'un sous-tableau de taille 1, nous devons doubler sa taille pour atteindre la taille $n$. En fait, il s'agit simplement d'une élévation la puissance : multiplier à plusieurs reprises par 2. Autrement dit, pour quelle valeur de $x$, $2^x$ atteint-il $n$ ? Si $n$ est une puissance exacte de 2, nous avons vu à la page 7 que la réponse est $\lg n$. Bien entendu, $n$ peut ne pas être une puissance exacte de 2, auquel cas la réponse sera entre 1 et $\lg n$. Enfin, nous remarquons que chaque itération de la boucle prend un temps constant. Autrement dit, le temps d'une seule itération ne dépend pas de la taille $n$ du tableau d'origine ni de la taille du sous-tableau examiné. Utilisons une notation asymptotique pour supprimer le facteur constant et les termes d'ordre inférieur. (Le nombre d'itérations de la boucle est-il égal à $\lg n$ ou

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

à $\lfloor \lg n \rfloor + 1$ ? Aucune importance.) Le temps d'exécution de la recherche dichotomique est donc $O(\lg n)$.

Nous utilisons ici la notation $O$ car nous voulons une déclaration qui couvre tous les cas. Dans le cas le plus défavorable, lorsque la valeur $x$ est absente du tableau, nous avons coupé en deux à plusieurs reprises, jusqu'à ce que le sous-tableau examiné soit vide. Nous obtenons alors un temps d'exécution $\Theta(\lg n)$. Dans le cas le plus favorable, lorsque $x$ est trouvé au cours de la première itération de la boucle, le temps d'exécution est $\Theta(1)$. Aucune notation $\Theta$ ne couvre tous les cas, mais un temps d'exécution $O(\lg n)$ est toujours exact pour la recherche dichotomique, tant que le tableau est initialement trié.

Il est possible d'obtenir mieux que le temps $\Theta(\lg n)$ du cas le plus défavorable, mais uniquement si les données sont organisées de manière plus élaborée et si certaines hypothèses sur les clés sont faites.

## Tri par sélection

Nous allons à présent nous intéresser au **tri** : réorganiser les éléments du tableau (ou **permuter** le tableau) afin que chaque élément soit inférieur ou égal à son successeur. Le premier algorithme de tri décrit, le tri par sélection, est à mon avis le plus simple car il est le premier auquel j'arrive lorsque j'ai besoin de concevoir un algorithme de tri. En revanche, il est loin d'être le plus rapide

Voici comment se déroule le tri par sélection sur les noms des auteurs des livres posés sur une étagère. Nous parcourons l'intégralité les livres sur l'étagère et recherchons celui dont le nom d'auteur arrive en premier dans l'ordre alphabétique. Supposons qu'il ait été écrit par Louisa May Alcott. (Si l'étagère comprend deux livres de cet auteur, ou plus, nous choisissons l'un d'eux.) Nous échangeons cet ouvrage avec celui de l'alvéole 1. L'ouvrage dans l'alvéole 1 est à présent celui d'un auteur dont le nom arrive en premier selon l'ordre alphabétique. Nous examinons de nouveau l'étagère, de gauche à droite, en commençant par l'ouvrage dans l'alvéole 2 afin de trouver le livre dans les emplacements 2 à $n$ dont le nom d'auteur arrive en premier dans l'alphabet. Supposons que cet auteur soit Jane Austen. Nous échangeons l'ouvrage avec celui de l'alvéole 2. Nous avons à présent dans les alvéoles 1 et 2 le premier et le deuxième livre dans l'ordre alphabétique. Nous répétons la procédure pour l'alvéole 3 et les suivantes. Après que nous avons placé le livre approprié dans l'alvéole $n - 1$ (peut-être est-il de H. G. Wells), la tâche est terminée car il reste un seul ouvrage (disons écrit par Oscar Wilde) et il se trouve dans l'alvéole $n$.

Pour convertir ce fonctionnement en un algorithme informatique, il suffit de remplacer l'étagère par un tableau et les ouvrages par des éléments du tableau.

Copyright © 2013 Dunod.

Voici le résultat :

*Procédure* TRI-SÉLECTION($A$, $n$)
*Entrées :*
- $A$ : un tableau.
- $n$ : le nombre d'éléments du tableau $A$ à trier.

*Résultat :* Les éléments de $A$ sont triés par ordre croissant.

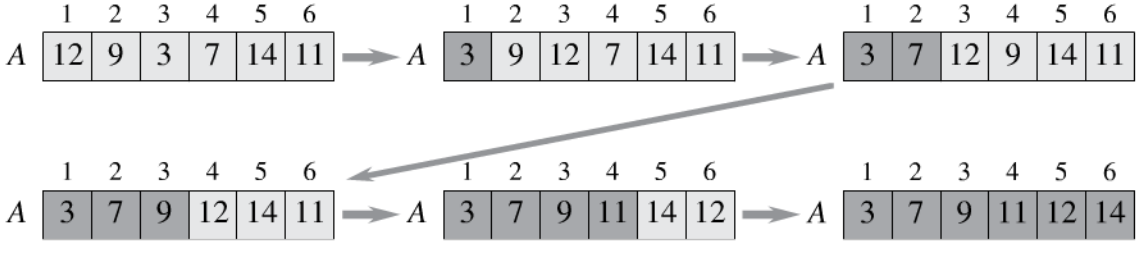
1. Pour $i = 1$ à $n - 1$ :
   A. Fixer *minimum* à l'indice du plus petit élément du sous-tableau $A[i..n]$.
   B. Échanger $A[i]$ et $A[minimum]$.

Pour trouver le plus petit élément, il suffit d'employer une variante de la recherche linéaire. Nous déclarons initialement que $A[i]$ est le plus petit élément rencontré jusqu'à présent dans le sous-tableau. Ensuite, nous parcourons les éléments restants du sous-tableau, en actualisant l'indice du plus petit élément chaque fois que nous trouvons un élément inférieur au minimum actuel. Voici la nouvelle version de la procédure :

*Procédure* TRI-SÉLECTION($A$, $n$)
*Entrées et résultat :* identiques aux précédents.

1. Pour $i = 1$ à $n - 1$ :
   A. Fixer *minimum* à 1.
   B. Pour $j = i + 1$ à $n$ :
      i. Si $A[j] < A[minimum]$, alors fixer *minimum* à $j$.
   C. Échanger $A[i]$ et $A[minimum]$.

Cette procédure comprend des boucles imbriquées : la boucle de l'étape 1B se trouve dans celle de l'étape 1. La boucle interne effectue toutes ses itérations pour chaque itération individuelle de la boucle externe. Notez que la valeur de départ de *j* dans la boucle interne dépend de la valeur actuelle de *i* dans la boucle externe. L'illustration suivante montre le fonctionnement du tri par sélection sur un tableau de six éléments :




Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Le tableau initial se trouve en haut à gauche et chaque étape le montre après une itération de la boucle externe. Les éléments en gris foncé correspondent au sous-tableau trié.

Si vous souhaitez employer un invariant de boucle pour montrer que la procédure TRI-SÉLECTION trie le tableau de façon correcte, vous devez en trouver un pour chaque boucle. Voici des invariants de boucle, mais, puisque cette procédure est simple, nous n'en développerons pas l'argumentation :

> Au début de chaque itération de la boucle de l'étape 1, le sous-tableau $A[1..i-1]$ contient les $i-1$ plus petits éléments du tableau $A$ complet et ils sont triés.
>
> Au début de chaque itération de la boucle de l'étape 1B, $A[minimum]$ est le plus petit élément dans le sous-tableau $A[i..j-1]$.

Quel est le temps d'exécution de TRI-SÉLECTION ? Nous allons montrer qu'il est $\Theta(n^2)$. Le point essentiel est d'analyser le nombre d'itérations effectuées par la boucle interne, en notant que chacune prend un temps $\Theta(1)$. (Les facteurs constants dans les bornes inférieure et supérieure de la notation $\Theta$ peuvent varier car l'affectation de *minimum* ne se produit pas nécessairement lors de chaque itération.) Calculons le nombre d'itérations en fonction de la valeur de la variable de boucle externe $i$. Lorsque $i$ est égal à 1, la boucle interne effectue $j$ itérations de 2 à $n$, c'est-à-dire $n-1$. Lorsque $i$ est égal à 2, la boucle interne réalise $j$ itérations de 3 à n, c'est-à-dire $n-2$. À chaque incrémentation de $i$ dans la boucle externe, la boucle interne effectue une itération de moins. De façon générale, la boucle interne s'exécute $n-i$ fois. Lors de la dernière itération de la boucle externe, lorsque $i$ est égal à $n-1$, la boucle interne réalise une seule itération. Par conséquent, le nombre total d'itérations de la boucle interne est calculé par la formule suivante :

$$(n-1)+(n-2)+(n-3)+\ldots+2+1$$

Cette somme correspond à une *suite arithmétique*, dont voici une propriété de base pour tout $k$ entier positif :

$$k+(k-1)+(k-2)+\ldots+2+1=\frac{k(k+1)}{2}$$

En remplaçant $k$ par $n-1$, nous voyons que le nombre total d'itérations de la boucle interne est égal à $(n-1)n\ /\ 2$, c'est-à-dire $(n^2-n)\ /\ 2$. Servons-nous de la notation asymptotique pour retirer le terme d'ordre inférieur $(-n)$ et le facteur constant $(1\ /\ 2)$. Nous pouvons alors dire que le nombre total d'itérations de la boucle interne est $\Theta(n^2)$. Le temps d'exécution de TRI-SÉLECTION est par conséquent $\Theta(n^2)$. Notez qu'il s'agit d'une déclaration générale qui couvre tous les cas. Quelles que soient les valeurs réelles des éléments, la boucle interne s'exécute à $\Theta(n^2)$ reprises.

Copyright © 2013 Dunod.

Il existe une autre manière de constater que le temps d'exécution est $\Theta(n^2)$, sans passer par la suite arithmétique. Nous allons montrer séparément que le temps d'exécution est à la fois $O(n^2)$ et $\Omega(n^2)$ ; en réunissant les bornes asymptotiques supérieure et inférieure, nous obtenons $\Theta(n^2)$. Pour déterminer que le temps d'exécution est $O(n^2)$, observons que chaque itération de la boucle externe exécute la boucle interne au moins $n - 1$ fois. Cela donne $O(n)$ car chaque itération de la boucle interne se fait en un temps constant. Puisque la boucle externe itère $n - 1$ fois, c'est-à-dire également $O(n)$, le temps total passé dans la boucle interne est $O(n)$ fois $O(n)$, c'est-à-dire $O(n^2)$. Pour déterminer que le temps d'exécution est $\Omega(n^2)$, observons que dans chacune des $n / 2$ premières itérations de la boucle externe, la boucle interne est exécutée au moins $n / 2$ fois, pour un total d'au moins $n / 2$ multiplié par $n / 2$, c'est-à-dire $n^2 / 4$ fois. Puisque chaque itération de la boucle interne se fait en un temps constant, nous voyons que le temps d'exécution est au moins une constante multipliée par $n^2 / 4$, ou $\Omega(n^2)$.

Il nous reste deux réflexions à propos du tri par sélection. Premièrement, son temps d'exécution asymptotique $\Theta(n^2)$ est pire que celui de tous des algorithmes de tri que nous allons étudier. Deuxièmement, si vous examinez attentivement son fonctionnement, vous verrez que le temps d'exécution $\Theta(n^2)$ vient des comparaisons effectuées à l'étape 1Bi. En revanche, le nombre de déplacements des éléments du tableau n'est que de $\Theta(n)$, car l'étape 1C est effectuée à seulement $n - 1$ reprises. Si le déplacement des éléments du tableau est particulièrement coûteux en temps, éventuellement parce qu'ils sont volumineux ou stockés sur un périphérique lent comme un disque, alors le tri par sélection peut se révéler intéressant.

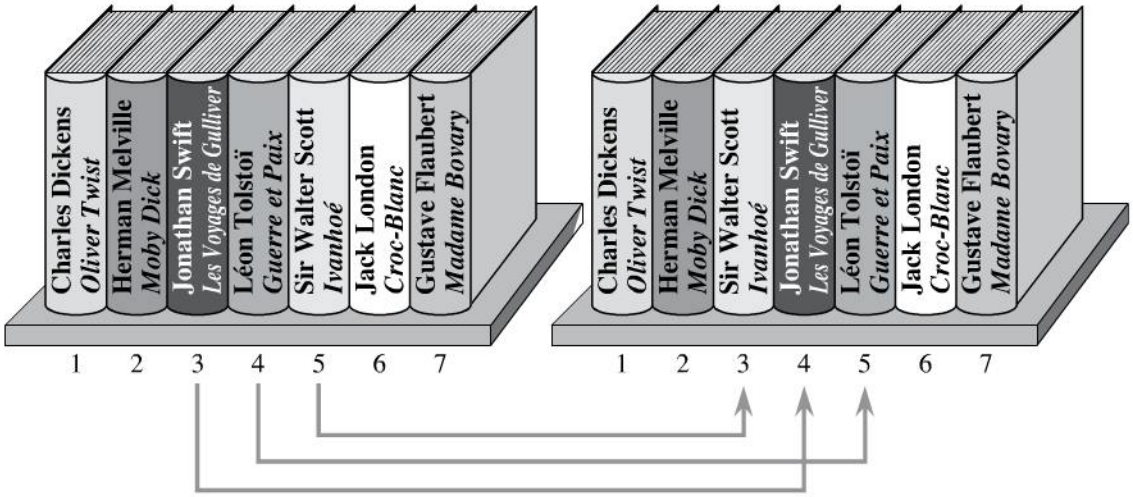
## Tri par insertion

Le tri par insertion diffère légèrement du tri par sélection, bien qu'ils soient tous deux dans la même veine. Dans le tri par sélection, lorsque nous avons décidé du livre à placer dans la *i*-ième alvéole, ceux des *i* premières alvéoles correspondaient aux *i* premiers ouvrages *globaux*, triés par ordre alphabétique sur le nom de l'auteur. Dans le tri par insertion, les ouvrages dans les *i* premières alvéoles seront ceux *qui se trouvaient à l'origine dans les* i *premières alvéoles*, mais à présent triés en fonction du nom de l'auteur.

Par exemple, supposons que les ouvrages dans les quatre premières alvéoles soient déjà triés par nom d'auteur ; ils ont été écrits par Charles Dickens, Herman Melville, Jonathan Swift et Léon Tolstoï. Supposons que l'ouvrage qui se trouve dans l'alvéole 5 soit de Sir Walter Scott. Dans le cas du tri par insertion, nous déplaçons les livres de Swift et de Tolstoï d'une alvéole vers la droite (ils passent des alvéoles 3 et 4 aux alvéoles 4 et 5) et plaçons ensuite le livre de Scott dans l'alvéole 3 vacante. Au moment où nous examinons le livre de Scott, nous ne nous préoccupons pas des ouvrages qui

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

se trouvent à sa droite (ceux de Jack London et Gustave Flaubert dans l'illustration ci-après) ; nous nous en occuperons ultérieurement.




Pour déplacer les livres de Swift et de Tolstoï, nous comparons tout d'abord le nom Tolstoï à Scott. Puisque Tolstoï vient après Scott, nous décalons cet ouvrage d'une alvéole vers la droite (de l'alvéole 4 à l'alvéole 5). Ensuite, nous comparons le nom d'auteur Swift à Scott, et décalons donc l'ouvrage de Swift d'une alvéole vers la droite (de l'alvéole 3 à l'alvéole 4), qui est devenue vacante suite au décalage du livre de Tolstoï. Puis nous comparons le nom d'auteur Melville à Scott. Cette fois-ci, Melville ne vient pas après Scott. Nous arrêtons donc la comparaison des noms d'auteurs car nous avons déterminé que l'ouvrage de Scott doit être placé à droite de celui de Melville et à gauche de celui de Swift. Nous le plaçons alors dans l'alvéole 3, qui est devenue vacante suite au décalage de l'ouvrage de Swift.

Dans la traduction de ce principe en un tri par insertion d'un tableau, le sous-tableau $A[1..i-1]$ contient uniquement les éléments qui se trouvaient à l'origine dans les $i-1$ emplacements du tableau et qui sont à présent triés. Pour déterminer où l'élément placé initialement en $A[i]$ doit aller, le tri par insertion parcourt $A[1..i-1]$ en commençant à $A[i-1]$ et en progressant vers la gauche. Il décale chaque élément supérieur à celui-ci d'une position vers la droite. Lorsqu'il trouve un élément qui n'est pas supérieur à $A[i]$ ou lorsque l'extrémité gauche du tableau est atteinte, l'élément à l'origine en $A[i]$ est déposé au nouvel emplacement dans le tableau.

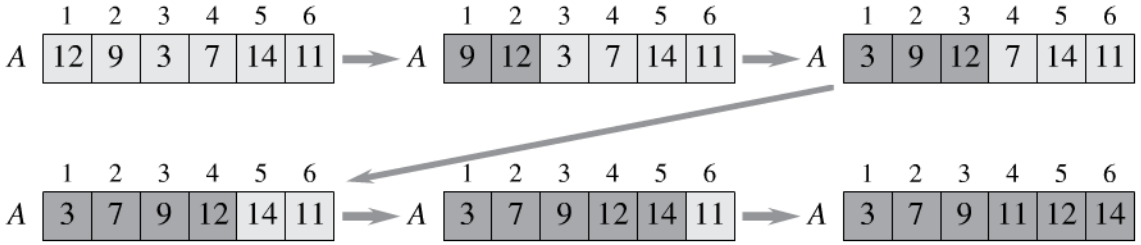
*Procédure* TRI-INSERTION$(A,n)$
*Entrées et résultat :* Identiques à ceux de TRI-SÉLECTION.
1. Pour $i = 2$ à $n$ :
   A. Fixer *clé* à $A[i]$ et $j$ à $i-1$.

Copyright © 2013 Dunod.

B. Tant que $j > 0$ et $A[j] > clé$, faire la chose suivante :
   i. Fixer $A[j + 1]$ à $A[j]$.
   ii. Décrémenter $j$ (c'est-à-dire fixer $j$ à $j - 1$).
C. Fixer $A[j + 1]$ à *clé*.

Le test effectué lors de l'étape 1B se fonde sur le **court-circuitage** de l'opérateur « et » : si l'expression placée à gauche, $j > 0$, est fausse, alors l'expression placée à droite $A[j] > clé$ n'est pas évaluée. S'il tente d'accéder à $A[j]$ alors que $j \leq 0$, une erreur d'indice de tableau est générée.

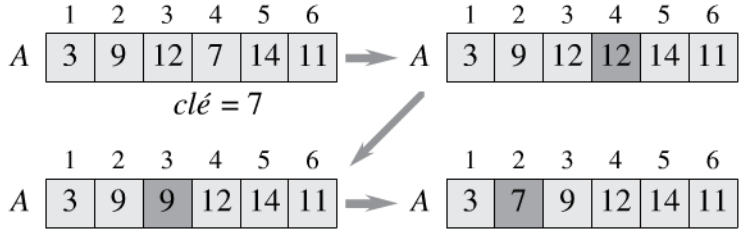
Voici le fonctionnement du tri par insertion sur le tableau utilisé à la page 35 pour le tri par sélection :




Le tableau de départ se trouve en haut à gauche et chaque étape l'affiche après une itération de la boucle externe de l'étape 1. Les éléments en gris foncé correspondent au sous-tableau trié. Voici l'invariant de boucle de la boucle externe (nous ne le prouverons pas) :

> Au début de chaque itération de la boucle de l'étape 1, le sous-tableau $A[1..i - 1]$ est constitué des éléments qui se trouvaient à l'origine dans $A[1..i - 1]$, mais qui sont à présent triés.

L'illustration suivante montre le fonctionnement de la boucle interne de l'étape 1B avec le tableau précédent lorsque $i$ est égal à 4. Nous supposons que le sous-tableau $A[1..3]$ contient les éléments à l'origine dans les trois premiers emplacements du tableau mais qui sont à présent triés. Pour déterminer où placer l'élément qui se trouvait initialement en $A[4]$, nous le mémorisons dans la variable *clé* et décalons ensuite chaque élément de $A[1..3]$ supérieur à *clé* d'une position vers la droite :




Les emplacements en gris foncé montrent la destination des éléments. Au cours de la dernière étape illustrée, la valeur de $A[1]$, 3, n'est pas supérieure

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

à la valeur de *clé*, 7, et la boucle interne se termine donc. La valeur de *clé* est déposée dans l'emplacement qui se trouve à droite de $A[1]$, comme le montre la dernière étape. À l'étape 1A nous devons mémoriser dans *clé* la valeur qui se trouve dans $A[i]$ car la première itération de la boucle interne écrase $A[i]$.

Il est également possible que la boucle interne se termine si le test $j > 0$ devient faux. Cela se produit lorsque la valeur de *clé* est inférieure à tous les éléments dans $A[1..i-1]$. Lorsque $j$ est égal à 0, chaque élément de $A[1..i-1]$ a été décalé vers la droite et l'étape 1C place dont *clé* dans $A[1]$, exactement là où nous le souhaitons.

L'analyse du temps d'exécution de TRI-INSERTION est un tantinet plus complexe que celle de TRI-SÉLECTION. LE NOMBRE D'ITÉRATIONS DE LA BOUCLE INTERNE DE LA PROCÉDURE TRI-SÉLECTION dépend uniquement de l'indice $i$ de la boucle externe et aucunement des éléments eux-mêmes. En revanche, dans le cas de la procédure TRI-INSERTION, le nombre d'itérations de la boucle interne dépend à la fois de l'indice $i$ de la boucle externe et des valeurs des éléments du tableau.

POUR TRI-INSERTION, LE CAS LE PLUS FAVORABLE EST RENCONTRÉ LORSQUE LA BOUCLE INTERNE NE RÉALISE AUCUNE ITÉRATION. POUR QUE CELA SE PRODUISE, LE TEST $A[j] > clé$ doit être faux dès la première fois pour chaque valeur de $i$. Autrement dit, il faut que $A[i-1] \leq A[i]$ lors de chaque exécution de l'étape 1B. Cette situation arrive uniquement si le tableau $A$ est déjà trié lorsque la procédure débute. Dans ce cas, la boucle externe itère à $n-1$ reprises et chaque itération se fait en un temps constant. TRI-INSERTION demande donc un temps $\Theta(n)$.

Le cas le plus défavorable se rencontre lorsque la boucle interne effectue à chaque fois le nombre maximum d'itérations possibles. Le test $A[j] > clé$ doit être à présent toujours vrai et la boucle doit se terminer car le test $j > 0$ est faux. Chaque élément $A[i]$ doit se déplacer vers la gauche tout le long du tableau. Cette situation arrive uniquement si le tableau $A$ est initialement trié en ordre inverse. Dans ce cas, pour chaque itération de la boucle externe, la boucle interne est exécutée à $i-1$ reprises. Puisque la boucle externe s'exécute avec $i$ allant de 2 à $n$, le nombre d'itérations de la boucle interne forme une suite arithmétique :

$$1 + 2 + 3 + ... + (n-2) + (n-1)$$

Nous l'avons vu pour le tri par sélection, nous obtenons $\Theta(n^2)$. Puisque chaque itération de la boucle interne se fait en une durée constante, le temps d'exécution dans le cas le plus défavorable pour le tri par insertion est $\Theta(n^2)$. Ainsi, dans le cas le plus défavorable, les tris par sélection et par insertion ont des temps d'exécution qui sont asymptotiquement identiques.

Est-il sensé d'essayer de comprendre ce qui se passe dans un cas normal avec le tri par insertion ? Cela dépend de la forme que revêt une entrée

Copyright © 2013 Dunod.

« normale ». Si l'ordre des éléments dans le tableau d'entrée est véritablement aléatoire, nous pouvons supposer que chaque élément est plus grand qu'environ la moitié des éléments qui le précèdent et plus petit que l'autre moitié. Par conséquent, chaque fois que la boucle interne s'exécute, elle réalise environ $(i - 1) / 2$ itérations. Le temps d'exécution est donc réduit de moitié par rapport au pire des cas. Mais 1 / 2 n'est qu'un facteur constant et, de façon asymptotique, le temps ne sera pas différent de celui du pire des cas, c'est-à-dire encore $\Theta(n^2)$.

Le tri par insertion est parfaitement adapté à des tableaux qui sont presque triés. Supposons que chaque élément du tableau se trouve initialement dans les $k$ positions où il doit arriver dans la version triée. Dans ce cas, le nombre total de déplacements d'un élément donné sur toutes les itérations de la boucle interne est au plus égal à $k$. Par conséquent, le nombre total de déplacements de tous les éléments sur toutes les itérations de la boucle interne est au plus égal à $kn$. Nous pouvons en conclure que le nombre total d'itérations de la boucle interne est au plus égal à $kn$, car chacune déplace exactement un élément d'une position. Si $k$ est une constante, alors le temps d'exécution total du tri par insertion est seulement $\Theta(n)$, car la notation $\Theta$ absorbe le facteur constant $k$. En réalité, nous pouvons même tolérer que certains éléments se déplacent sur une plus longue distance dans le tableau, tant que leur nombre reste faible. Notamment, si $l$ éléments peuvent se déplacer n'importe où dans le tableau (de sorte que chacun d'eux puisse se déplacer d'au plus $n - 1$ positions) et si les $n - l$ éléments restants peuvent se déplacer vers au plus $k$ positions, alors le nombre maximum de décalages est $l(n - 1) + (n - l)k = (k + l)n - (k + 1)l$, c'est-à-dire $\Theta(n)$ si $k$ et $l$ sont constants.

Si nous comparons les temps d'exécution asymptotiques du tri par insertion et du tri par sélection, nous constatons que, dans le cas le plus défavorable, ils sont identiques. Le tri par insertion est plus efficace si le tableau est déjà pratiquement trié. En revanche, le tri par sélection présente toutefois un avantage : le tri par sélection déplace les éléments à $\Theta(n)$ reprises quels qu'ils soient, tandis que le tri par insertion peut déplacer des éléments jusqu'à $\Theta(n^2)$ fois car chaque exécution de l'étape 1Bi de Tri-Insertion déplace un élément. Nous l'avons mentionné à la page 37 pour le tri par sélection, si le déplacement d'un élément est particulièrement coûteux en temps et si vous n'avez aucune raison de penser que l'entrée fournie permette de s'approcher du cas le plus favorable, alors il est préférable de choisir le tri par sélection plutôt que celui par insertion.

## Tri par fusion

L'algorithme de tri suivant, le tri par fusion, affiche un temps d'exécution de seulement $\Theta(n \lg n)$ dans tous les cas. Si l'on compare ce temps d'exécution avec celui de $\Theta(n^2)$ dans le cas le plus défavorable du tri par sélection

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

et du tri par insertion, nous remplaçons un facteur $n$ par un facteur $\lg n$. Nous l'avons mentionné à la page 7 du Chapitre 1, c'est un troc que nous devons accepter systématiquement.

Le tri par fusion a toutefois deux inconvénients par rapport aux deux autres algorithmes de tri étudiés. Premièrement, le facteur constant masqué par la notation asymptotique est plus élevé que celui des deux premiers algorithmes. Toutefois, dès que le tableau est suffisamment grand, cela n'a pas vraiment d'importance. Deuxièmement, le tri par fusion ne travaille pas sur le tableau d'origine : il doit réaliser des copies de l'intégralité du tableau d'entrée. Cette copie de toutes les entrées est à comparer à la copie d'une seule entrée du tableau dans les tris par sélection et par insertion. Si la quantité de mémoire utilisée est une contrainte, le tri par fusion pourrait ne pas convenir.

Le tri par fusion met en œuvre une technique algorithmique répandue baptisée **diviser pour régner**. Elle consiste à décomposer le problème d'origine en sous-problèmes comparables, à résoudre ces sous-problèmes de manière récursive et à combiner les solutions obtenues de façon à résoudre le problème initial. Au Chapitre 2, nous avons vu que la récursion exigeait que chaque appel récursif se fasse sur une instance plus petite du même problème et qu'il fallait arriver à un cas de base. Voici une description générale d'un algorithme de type diviser-pour-régner :

1. **Diviser** le problème en plusieurs sous-problèmes qui représentent des instances plus petites du même problème.
2. **Régner** sur les sous-problèmes en les résolvant de façon récursive. S'ils sont suffisamment petits, les résoudre comme des cas de base.
3. **Combiner** les solutions trouvées aux sous-problèmes afin de construire la solution du problème initial.

Lorsque nous trions les ouvrages sur l'étagère à l'aide du tri par fusion, chaque sous-problème consiste à trier les livres placés dans des alvéoles consécutives. Initialement, nous voulons trier les $n$ ouvrages dans les alvéoles 1 à $n$. Dans un sous-problème général, nous trions tous les livres des alvéoles $p$ à $r$. Voici comment nous appliquons la technique diviser-pour-régner :

1. **Diviser** en déterminant le nombre $q$ de l'alvéole qui se trouve à mi-chemin entre $p$ et $r$. Nous employons la méthode déjà mise en œuvre pour trouver le point central dans la recherche dichotomique : additionner $p$ et $q$, diviser le résultat par 2, et prendre la partie entière.
2. **Régner** en triant récursivement les ouvrages dans chacun des deux sous-problèmes créés lors de l'étape de division : trier récursivement les livres qui se trouvent dans les alvéoles $p$ à $q$, et ceux qui se trouvent dans les alvéoles $q + 1$ à $r$.
3. **Combiner** par fusion des livres triés des alvéoles $p$ à $q$ et des alvéoles $q + 1$ à $r$, de sorte que tous les ouvrages dans les alvéoles $p$ à $r$ soient triés. Nous verrons plus loin comment effectuer cette fusion.

Copyright © 2013 Dunod.

Le cas de base correspond au tri de moins de deux livres (c'est-à-dire lorsque $p \geq r$), car une collection d'aucun ouvrage ou d'un seul ouvrage est évidemment déjà triée.

Pour passer de ce principe au tri d'un tableau, les ouvrages dans les alvéoles $p$ à $r$ correspondent au sous-tableau $A[p..r]$. Voici la procédure de tri par fusion, qui appelle une procédure FUSIONNER$(A,p,q,r)$ de façon à combiner les sous-tableaux triés $A[p..q]$ et $A[q+1..r]$ en un seul sous-tableau $A[p..r]$ trié :

*Procédure* TRI-FUSION$(A,p,r)$

*Entrées :*

- $A$ : un tableau.
- $p$, $r$ : les indices de début et de fin d'un sous-tableau de $A$.

*Résultat :* Les éléments du sous-tableau $A[p..r]$ sont triés par ordre croissant.

1. Si $p \geq r$, alors le sous-tableau $A[p..r]$ a au plus un élément et il est déjà trié. Retourner simplement sans plus de traitements.
2. Sinon faire la chose suivante :
   A. Fixer $q$ à $\lfloor (p+r) / 2 \rfloor$.
   B. Appeler récursivement TRI-FUSION$(A,p,q)$.
   C. Appeler récursivement TRI-FUSION$(A,q+1,r)$.
   D. Appeler FUSIONNER$(A,p,q,r)$.

Nous n'avons pas encore présenté le fonctionnement de la procédure FUSIONNER, mais nous pouvons néanmoins illustrer celui de TRI-FUSION. Nous partons du tableau suivant :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 3 | 7 | 14 | 11 | 6 | 2 | 10 | 5 |

L'appel initial est TRI-FUSION$(A,1,10)$. L'étape 2A détermine que la valeur de $q$ est 5. Les appels récursifs aux étapes 2B et 2C sont donc TRI-FUSION$(A,1,5)$ et TRI-FUSION$(A,6,10)$ :

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 12 | 9 | 3 | 7 | 14 |

| 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|
| 11 | 6 | 2 | 10 | 5 |

Après que les deux appels récursifs ont retourné, les deux sous-tableaux correspondants sont triés :

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 3 | 7 | 9 | 12 | 14 |

| 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|
| 2 | 5 | 6 | 10 | 11 |

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

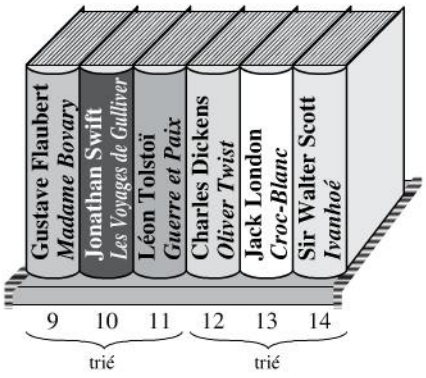
Enfin, l'appel FUSIONNER($A$,1,5,10) à l'étape 2D fusionne les deux sous-tableaux triés en un seul sous-tableau trié, qui, dans ce cas, correspond au tableau entier :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 7 | 9 | 10 | 11 | 12 | 14 |

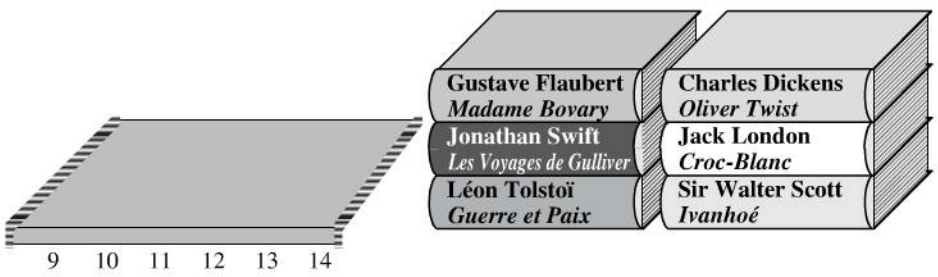
Si nous détaillons la récursion, nous obtenons le schéma de la page 45. Les flèches divergentes indiquent les étapes de division, tandis que les flèches convergentes représentent celles de fusion. Les variables $p$, $q$ et $r$ qui apparaissent au-dessus de chaque sous-tableau sont placées aux indices correspondants dans chaque appel récursif. Les noms en italique donnent l'ordre dans lequel les appels de la procédure sont effectués après l'appel initial TRI-FUSION($A$,1,10). Par exemple, l'appel FUSIONNER($A$,1,3,5) est le treizième appel de procédure après l'appel initial, et l'appel TRI-FUSION ($A$,6,7) est le seizième.

Le véritable travail se passe dans la procédure FUSIONNER. Elle doit donc non seulement avoir le comportement juste, mais elle doit également être rapide. Si nous fusionnons un total de $n$ éléments, le meilleur que nous puissions espérer est un temps $\Theta(n)$, car chaque élément doit arriver à l'emplacement approprié et nous pouvons effectuer une fusion en un temps linéaire.

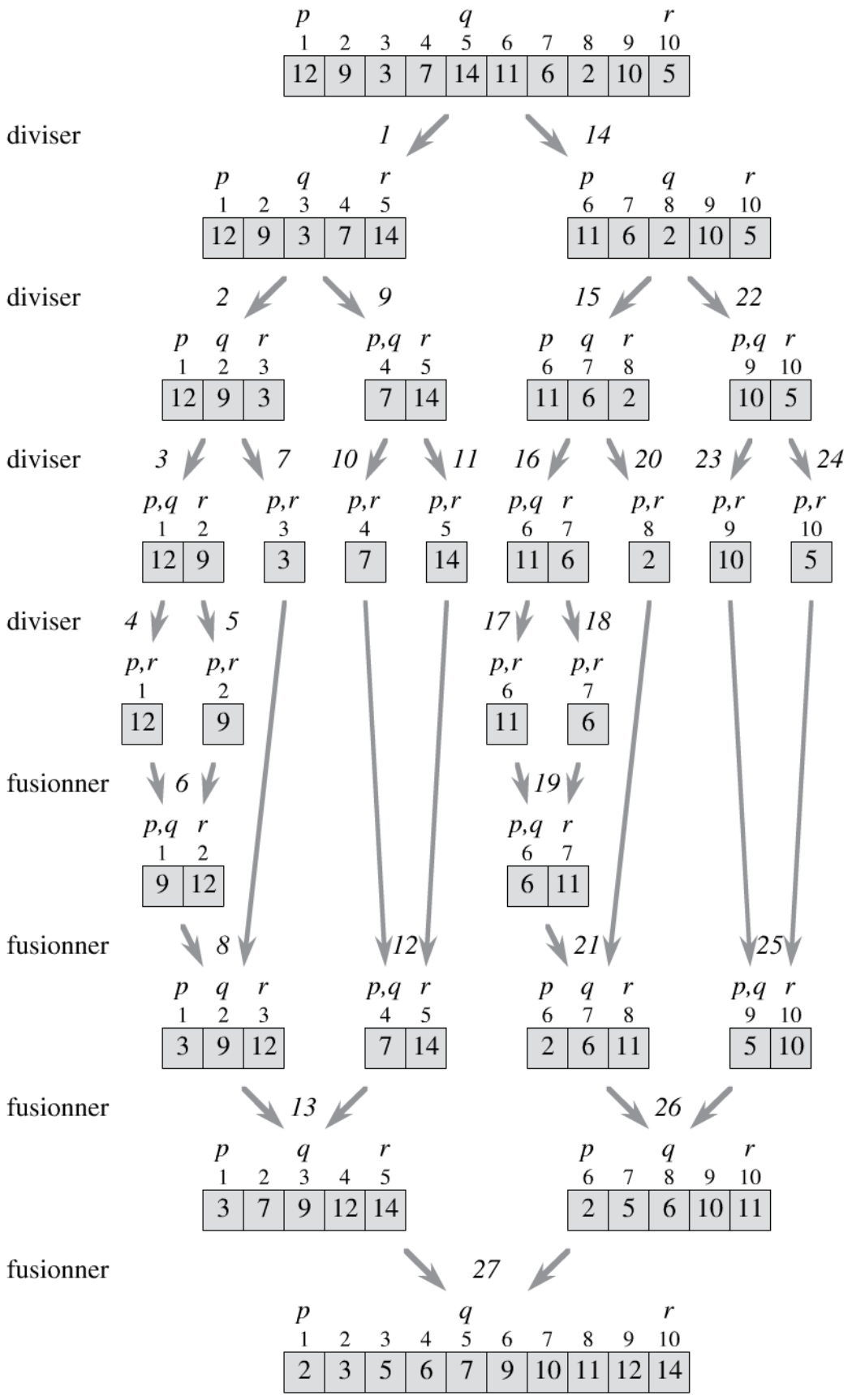
Reprenons notre exemple des livres et examinons uniquement la partie de l'étagère qui va de l'alvéole 9 à l'alvéole 14. Supposons que nous ayons trié les ouvrages des alvéoles 9 à 11, ainsi que ceux des alvéoles 12 à 14 :




Nous constituons une pile à partir des livres des alvéoles 9 à 11 et une autre avec ceux des alvéoles 12 à 14, en plaçant au sommet d'une pile l'ouvrage dont le nom de l'auteur arrive en premier par ordre alphabétique :



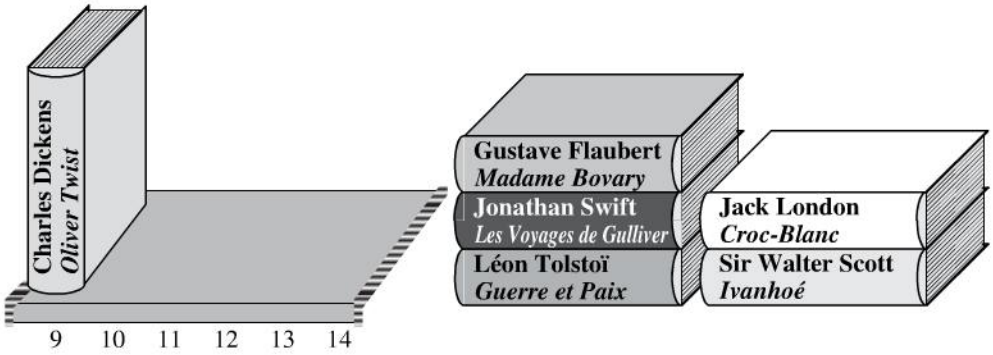

Copyright © 2013 Dunod.



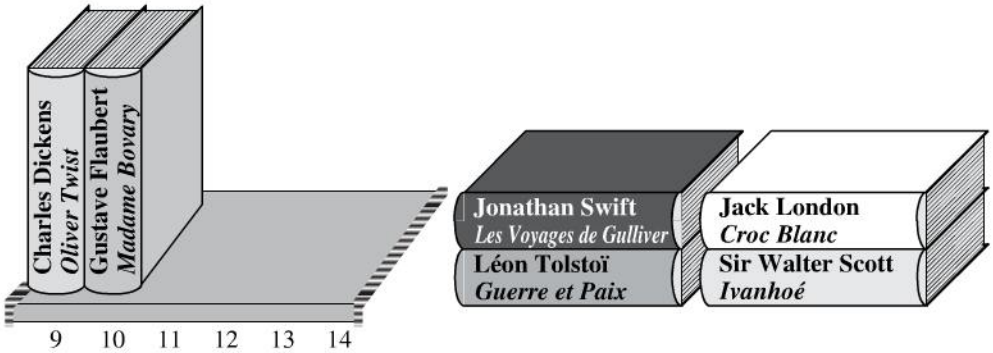

Puisque les piles sont déjà triées, le livre que nous devons remettre dans l'alvéole 9 est celui qui se trouve au sommet de l'une des deux piles : soit celui de Gustave Flaubert, soit celui de Charles Dickens. Nous déterminons

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

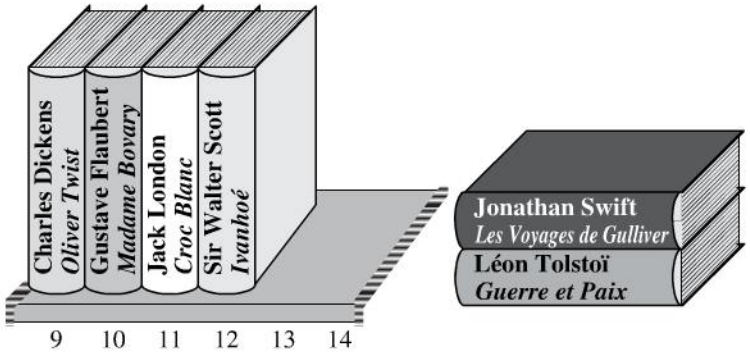
que le livre de Dickens vient avant celui de Flaubert et nous le déplaçons donc dans l'alvéole 9 :




Ensuite, l'ouvrage qui doit aller dans l'alvéole 10 est soit celui au sommet de la première pile (écrit par Flaubert), soit celui au sommet de la seconde pile (écrit par Jack London). Nous déplaçons le livre de Flaubert dans l'alvéole 10 :

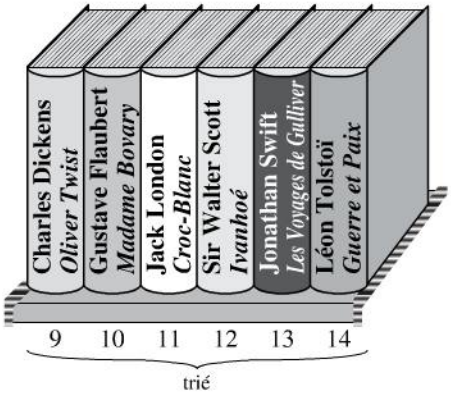



Nous comparons les ouvrages qui se trouvent au sommet de chaque pile, ceux de Jonathan Swift et de Jack London, et déplaçons ce dernier dans l'alvéole 11. Il nous reste le livre de Sir Walter Scott en haut de la pile de droite et, puisqu'il est face à l'ouvrage de Swift, nous le déplaçons dans l'alvéole 12. À ce stade, la pile de droite est vide :




Il ne nous reste plus qu'à déplacer dans l'ordre les ouvrages de la pile de gauche dans les alvéoles restantes. Tous les ouvrages des alvéoles 9 à 14 sont à présent triés :

Copyright © 2013 Dunod.




Cette procédure de fusion est-elle efficace ? Nous déplaçons chaque ouvrage à deux reprises : la première pour le retirer de l'étagère et le placer dans une pile, la seconde pour le déplacer du sommet de la pile et le remettre sur l'étagère. Par ailleurs, pour décider du livre à replacer sur l'étagère, nous devons comparer uniquement les deux ouvrages situés au sommet des piles. Par conséquent, pour fusionner $n$ livres, nous effectuons $2n$ déplacements et $n$ comparaisons de deux ouvrages.

Pourquoi devons-nous enlever les ouvrages de l'étagère ? Et si nous les laissions en place en nous contentant de mémoriser ceux que nous avons placés dans les alvéoles appropriées et les autres ? En réalité, cette approche exige un travail beaucoup plus important. Par exemple, supposons que chaque ouvrage de la moitié droite doive venir avant chaque ouvrage de la moitié gauche. Avant que nous puissions déplacer le premier livre de la moitié droite vers la première alvéole de la moitié gauche, nous devrions décaler d'une alvéole vers la droite chaque livre de la moitié gauche afin de faire de la place. La même opération devra être répétée pour le livre suivant de la moitié droite. Nous devrions donc décaler la moitié des ouvrages (tous ceux qui se trouvaient au départ dans la moitié gauche) chaque fois que nous voudrions placer un ouvrage de la moitié droite dans l'alvéole appropriée.

Voilà pourquoi la fusion ne se fait pas sur place[1]. Revenons à présent à la fusion des sous-tableaux triés $A[p..q]$ et $A[q+1..r]$ afin d'obtenir le sous-tableau $A[p..r]$. Nous commençons par copier les éléments à fusionner depuis le tableau $A$ vers des tableaux temporaires, que nous fusionnons ensuite dans $A$. Soit $n_1 = q - p + 1$ le nombre d'éléments de $A[p..q]$ et $n_2 = r - q$ le nombre d'éléments de $A[q+1..r]$. Nous créons un tableau $B$ temporaire de $n_1$ éléments et un tableau $C$ temporaire de $n_2$ éléments. Les éléments de $A[p..q]$ sont copiés dans l'ordre dans $B$, ceux de $A[q+1..r]$ dans $C$. Nous pouvons ensuite remettre les éléments dans $A[p..q]$ sans craindre d'écraser les copies effectuées.

1. En réalité, il est possible d'effectuer une fusion sur place en un temps linéaire, mais la méthode est plutôt complexe.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

La fusion des éléments du tableau reprend la méthode employée avec les livres. Nous copions des éléments depuis les tableaux $B$ et $C$ vers le sous-tableau $A[p..r]$, tout en gérant des indices afin de suivre les plus petits éléments de $B$ et de $C$ qui n'ont pas encore été recopiés et de déplacer le plus petit des deux. En un temps constant, nous déterminons le plus petit élément, le recopions à l'emplacement correct dans $A[p..r]$ et actualisons les indices.

À un moment donné, tous les éléments de l'un des deux tableaux auront été recopiés dans $A[p..r]$. Cela correspond au moment où il ne reste plus qu'une seule pile de livres. Nous employons toutefois une astuce pour éviter de vérifier à chaque fois si l'un des tableaux est vide : à l'extrémité droite des tableaux $B$ et $C$ nous plaçons un élément supplémentaire supérieur à tous les autres. Le principe est comparable à celui de la sentinelle utilisée au Chapitre 2 dans la procédure RECHERCHE-LINÉAIRE-SENTINELLE. Dans le cas présent, nous utilisons $\infty$ (l'infini) comme clé de tri sentinelle. Ainsi, dès qu'un élément dont la clé de tri $\infty$ est le dernier plus petit élément de son tableau, il est certain de « perdre » lorsque l'on cherche à déterminer le tableau qui possède le plus petit élément restant[1]. Après que tous les éléments des tableaux $B$ et $C$ ont été recopiés, il ne reste plus que l'élément sentinelle dans chacun d'eux. Il est inutile de comparer les sentinelles car nous avons à ce moment-là copié tous les éléments réels dans $A[p..r]$. Puisque nous savons à l'avance que nous allons copier les éléments dans $A[p]$ à $A[r]$, nous pouvons arrêter la procédure dès que nous copions un élément dans $A[r]$. Nous exécutons donc une boucle avec un indice sur $A$ qui va de $p$ à $r$.

Voici la procédure FUSIONNER. Elle peut sembler longue, mais elle respecte la méthode décrite précédemment.

*Procédure* FUSIONNER$(A, p, q, r)$

*Entrées :*

- $A$ : un tableau.
- $p, q, r$ : indices dans $A$. On suppose que les sous-tableaux $A[p..q]$ et $A[q+1..r]$ sont déjà triés.

*Résultat :* Le sous-tableau $A[p..r]$ contient les éléments qui se trouvaient à l'origine dans $A[p..q]$ et $A[q+1..r]$, mais l'intégralité du sous-tableau $A[p..r]$ est à présent triée.

1. Fixer $n_1$ à $q - p + 1$ et fixer $n_2$ à $r - q$.

1. Dans la pratique, $\infty$ est représenté par une valeur qui est toujours supérieure à n'importe quelle clé de tri. Par exemple, si la clé est le nom de l'auteur, nous pouvons choisir ZZZZ pour $\infty$, à condition évidemment qu'aucun auteur réel ne se nomme ainsi.

Copyright © 2013 Dunod.

2. Créer les tableaux $B[1..n_1 + 1]$ et $C[1..n_2 + 1]$.
3. Copier $A[p..q]$ dans $B[1..n_1]$ et copier $A[q + 1..r]$ dans $C[1..n_2]$.
4. Fixer $B[n_1 + 1]$ et $C[n_2 + 1]$ à $\infty$.
5. Fixer $i$ et $j$ à 1.
6. Pour $k = p$ à $r$ :
   A. Si $B[i] \leq C[j]$, alors fixer $A[k]$ à $B[i]$ et incrémenter $i$.
   B. Sinon ($B[i] > C[j]$) fixer $A[k]$ à $C[j]$ et incrémenter $j$.

Les étapes 1 à 4 allouent les tableaux $B$ et $C$, copient $A[p..q]$ dans $B$ et $A[q + 1..r]$ dans $C$, et insèrent les sentinelles dans ces tableaux. Ensuite, chaque itération de la boucle principale à l'étape 6 recopie le plus petit élément restant à l'emplacement suivant dans $A[p..r]$. Elle se termine lorsqu'elle a recopié tous les éléments de $B$ et de $C$. Dans cette boucle, $i$ est un indice sur le plus petit élément restant dans $B$, $j$ jouant le même rôle pour $C$. $k$ désigne l'emplacement dans $A$ où l'élément doit être recopié.

Si nous fusionnons $n$ éléments (de sorte que $n = n_1 + n_2$), la copie des éléments dans les tableaux $B$ et $C$ prend un temps $\Theta(n)$, et la recopie de chaque élément dans $A[p..r]$ se fait en un temps constant. Le temps total de fusion est donc seulement $\Theta(n)$.

Nous avons déclaré précédemment que le temps d'exécution de l'intégralité de l'algorithme de tri par fusion est $\Theta(n \lg n)$. Nous allons faire l'hypothèse que la taille $n$ du tableau est une puissance de 2 afin que chaque division du tableau produise deux sous-tableaux de tailles égales. (Rien ne garantit que $n$ sera une puissance de 2 et les tailles des sous-tableaux pourront ne pas être égales lors d'un appel récursif donné. Une analyse rigoureuse tiendra compte de cette particularité, mais nous n'allons pas nous en préoccuper.)

Voici notre analyse du tri par fusion. Supposons que le tri d'un sous-tableau de $n$ éléments se fasse en un temps $T(n)$, une fonction qui augmente avec $n$ (puisque l'on peut raisonnablement penser qu'il faut plus longtemps pour trier un plus grand nombre d'éléments). La durée $T(n)$ est constituée des trois composantes du paradigme diviser-pour-régner, dont chaque temps d'exécution s'additionne :

1. Diviser se fait en un temps constant car cette opération consiste simplement à calculer l'indice $q$.
2. Régner implique les deux appels récursifs sur les sous-tableaux, chacun avec $n / 2$ éléments. Conformément à notre définition du temps de tri d'un sous-tableau, chaque appel récursif prend un temps $T(n / 2)$.
3. Combiner les résultats des deux appels récursifs par la fusion des sous-tableaux triés se fait en $\Theta(n)$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Puisque le temps constant de la division représente un terme d'ordre inférieur par rapport au temps $\Theta(n)$ mis pour combiner les résultats, nous pouvons l'intégrer à celui-ci et dire que le temps des deux opérations réunies est $\Theta(n)$. Régner coûte $T(n / 2) + T(n / 2)$, c'est-à-dire $2T(n / 2)$. Voici donc l'équation de $T(n)$ :

$$T(n) = 2T(n / 2) + f(n)$$

$f(n)$ représente le temps mis pour diviser et combiner, qui, comme nous l'avons indiqué, est $\Theta(n)$. Lors de l'étude des algorithmes, une pratique courante consiste à placer la notation asymptotique dans l'équation et d'en faire une fonction dont le nom importe peu. Nous récrivons donc l'équation sous la forme suivante :

$$T(n) = 2T(n / 2) + \Theta(n)$$

Vous avez certainement remarqué que quelque chose cloche. Nous avons défini la fonction $T$, qui décrit le temps d'exécution du tri par fusion, en utilisant cette même fonction. Il s'agit d'une **équation de récurrence** ou simplement **récurrence** Le problème est que nous voulons exprimer $T(n)$ de façon non récursive, c'est-à-dire non en terme de lui-même. La transformation d'une fonction exprimée sous forme d'une récurrence en une version non récursive peut poser des difficultés, mais pour un grand nombre d'équations de récurrence, nous pouvons appliquer une recette appelée « méthode générale ». Elle s'applique à de nombreuses équations de récurrence de la forme $T(n) = aT(n / b) + f(n)$, où $a$ et $b$ sont des constantes entières positives. Par chance, elle s'applique à l'équation de récurrence de notre tri par fusion et nous obtenons pour résultat que $T(n)$ est $\Theta(n \lg n)$.

Le temps d'exécution $\Theta(n \lg n)$ s'applique à tous les cas du tri par fusion, du meilleur au pire, en passant par tous les cas intermédiaires. Chaque élément est copié $\Theta(n \lg n)$ fois. En examinant la méthode FUSIONNER, vous pouvez voir que, lorsqu'elle est invoquée avec $p = 1$ et $r = n$, elle crée des copies des $n$ éléments et le tri par fusion n'opère donc vraiment pas sur place.

Copyright © 2013 Dunod.

## TRI RAPIDE

À l'instar du tri par fusion, le tri rapide se fonde sur le paradigme diviser-pour-régner, et donc la récursion. Toutefois, son utilisation de ce paradigme varie légèrement de celle du tri par fusion et il montre d'autres différences significatives :

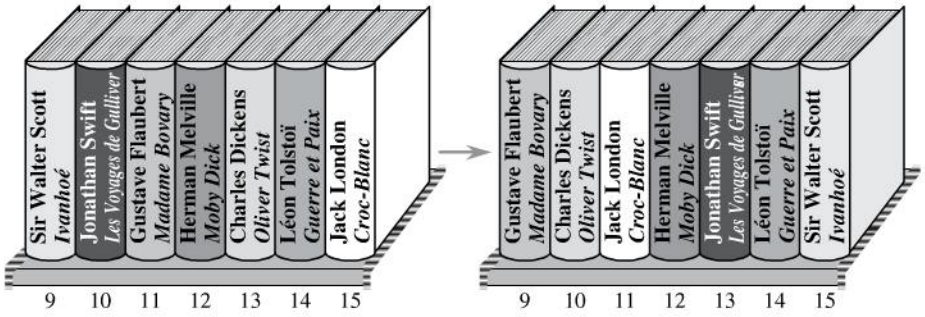
- Le tri rapide opère sur place.
- Le temps d'exécution asymptotique du tri rapide diffère entre le cas le plus défavorable et le cas normal. En particulier, le temps d'exécution pour le cas le plus défavorable est $\Theta(n^2)$, mais son temps d'exécution moyen est meilleur : $\Theta(n \lg n)$.

Le tri rapide possède également de bons facteurs constants (meilleurs que ceux du tri par fusion) et, dans la pratique, il se révèle souvent un bon algorithme de tri.

Voici comment le tri rapide utilise le paradigme diviser-pour-régner. Reprenons à nouveau notre étagère de livres à trier. Comme dans le cas du tri par fusion, nous voulons initialement trier les $n$ ouvrages placés dans les alvéoles 1 à $n$ et nous allons considérer le problème général du tri des livres dans les alvéoles $p$ à $r$.

1. **Diviser** en commençant par choisir n'importe quel livre qui se trouve dans les alvéoles $p$ à $r$. Nous appelons cet ouvrage le **pivot**. Nous réorganisons les livres sur l'étagère de sorte que tous les ouvrages dont les noms d'auteurs viennent avant celui du pivot ou qui ont été écrits par le même auteur se trouvent à gauche du pivot, tandis que tous ceux dont les noms d'auteurs viennent après celui du pivot se trouvent à sa droite.

   Dans cet exemple, nous choisissons comme pivot le livre à l'extrême droite, écrit Jack London, pour la réorganisation des ouvrages dans les alvéoles 9 à 15 :




   Après la réorganisation, que nous appelons **partitionnement** dans le tri rapide, les ouvrages de Flaubert et de Dickens, qui viennent avant London selon l'ordre alphabétique, sont sur la gauche du livre de London, et tous les autres ouvrages, dont les auteurs viennent après London par ordre alphabétique, sont sur sa droite. Vous remarquerez qu'après le partitionnement, les livres à gauche de celui écrit par London ne sont pas dans un ordre particulier, et c'est également le cas de ceux placés à sa droite.

2. **Régner** en triant récursivement les ouvrages situés à gauche et à droite du pivot. Autrement dit, si l'étape de division a déplacé le pivot vers l'alvéole $q$ (l'alvéole 11 dans l'exemple précédent), alors le tri récursif concerne les ouvrages des alvéoles $p$ à $q - 1$ et ceux des alvéoles $q + 1$ à $r$.
3. **Combiner**, en ne faisant rien ! Lorsque l'étape de règne a terminé le tri, la procédure est achevée. En effet, tous les ouvrages à gauche du pivot (dans les alvéoles $p$ à $q - 1$) viennent avant le pivot ou ont le même auteur et sont donc triés. De même, tous les ouvrages à droite du pivot (dans les alvéoles $q + 1$ à $r$) viennent après le pivot et sont donc triés. Les ouvrages dans les alvéoles $p$ à $r$ ne peuvent pas s'empêcher d'être triés !

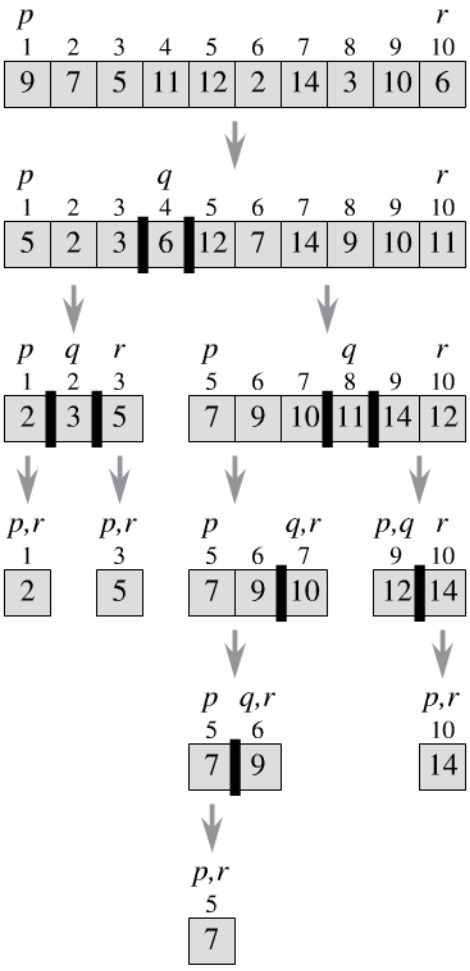
Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Si vous remplacez l'étagère par un tableau et les livres par des éléments du tableau, vous obtenez la stratégie du tri rapide. À l'instar du tri par fusion, le cas de base se rencontre lorsque le sous-tableau à trier comprend moins de deux éléments.

La procédure du tri rapide suppose l'invocation d'une procédure PARTITIONNER$(A, p, r)$, qui se charge du partitionnement du sous-tableau $A[p..r]$ et retourne l'indice $q$ du pivot.

*Procédure* TRI-RAPIDE$(A, p, r)$
*Entrées et résultat :* identiques à ceux de TRI-FUSION.
1. Si $p \geq r$, alors retourner simplement sans autre traitement.
2. Sinon faire la chose suivante :
   A. Appeler PARTITIONNER$(A, p, r)$ et affecter le résultat à $q$.
   B. Appeler récursivement TRI-RAPIDE$(A, p, q-1)$.
   C. Appeler récursivement TRI-RAPIDE$(A, q+1, r)$.

L'appel initial est TRI-RAPIDE$(A, 1, n)$, semblable à celui de la procédure TRI-FUSION. La figure suivante illustre un exemple de déroulement de la récursion, avec les indices $p$, $q$ et $r$ donnés pour chaque sous-tableau dans lequel $p \leq r$ :




Copyright © 2013 Dunod.

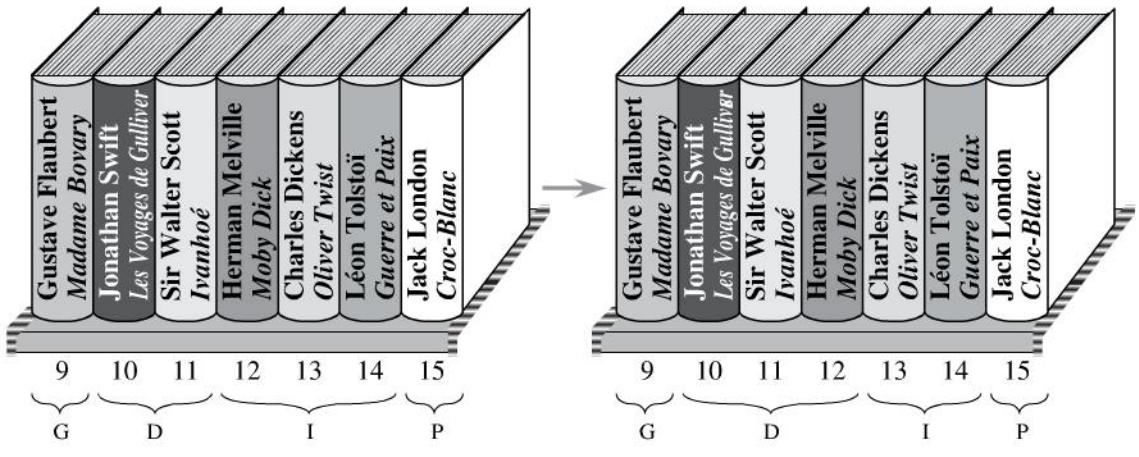
La valeur la plus basse dans chaque emplacement de tableau donne le dernier élément qui a été mis à cet endroit. Si vous lisez le tableau de gauche à droite, en examinant la valeur la plus basse dans chaque emplacement, vous constatez que le tableau est trié.

Le point clé du tri rapide réside dans le partitionnement. Tout comme nous sommes capables de fusionner $n$ éléments en un temps $\Theta(n)$, nous pouvons partitionner $n$ éléments en un temps $\Theta(n)$. Voici comment se passe le partitionnement des ouvrages qui se trouvent dans les alvéoles $p$ à $r$ sur l'étagère. Nous choisissons comme pivot l'ouvrage qui se trouve à l'extrême droite (celui dans l'alvéole $r$). À un moment donné, chaque ouvrage se trouvera dans un des quatre groupes suivants, qui vont des alvéoles $p$ à $r$, de gauche à droite :

- **Groupe G** (groupe de gauche) : les livres dont les auteurs viennent avant l'auteur du pivot selon l'ordre alphabétique ou qui ont le même auteur.
- **Groupe D** (groupe de droite) : les livres dont les auteurs viennent après l'auteur du pivot selon l'ordre alphabétique.
- **Groupe I** (groupe des inconnus) : les livres qui n'ont pas encore été examinés et pour lesquels nous ne savons pas si les auteurs sont avant ou après celui du pivot.
- **Groupe P** (pivot) : un seul livre, le pivot.

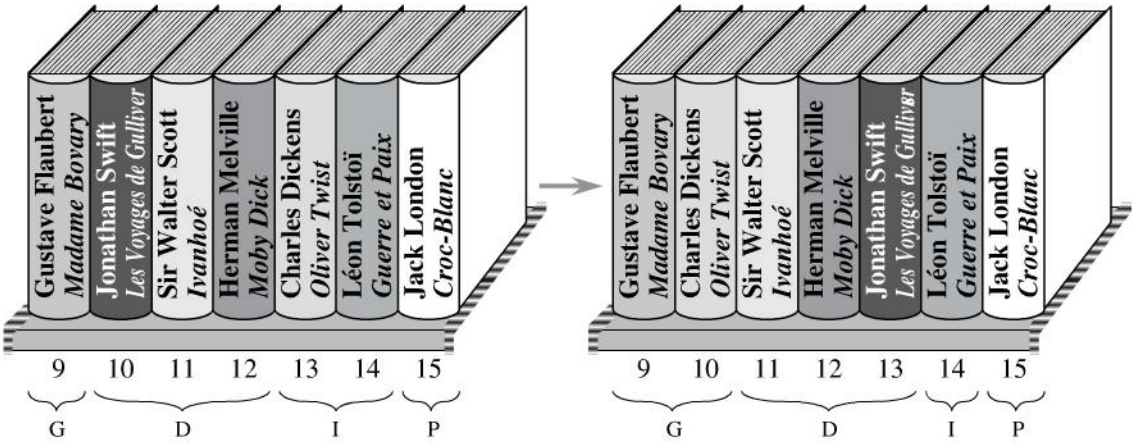
Nous parcourons de gauche à droite les ouvrages du groupe I. Chacun est comparé au pivot et déplacé dans le groupe G ou le groupe D. Nous nous arrêtons lorsque nous arrivons au pivot. L'ouvrage comparé au pivot est toujours le livre le plus à gauche du groupe I.

- Si l'auteur du livre se trouve après l'auteur du pivot, alors il devient le livre le plus à droite du groupe D. Puisqu'il était le plus à gauche du groupe I et que ce groupe suit immédiatement le groupe D, il suffit de déplacer la ligne de séparation des groupes D et I d'une alvéole vers la droite, sans toucher aux ouvrages :




© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

- Si l'auteur du livre se trouve avant l'auteur du pivot ou s'il est identique, alors nous en faisons le livre le plus à droite du groupe G. Nous l'échangeons avec le livre le plus à gauche du groupe D et déplaçons la ligne de séparation des groupes G et D et celle des groupes D et I d'une alvéole vers la droite :




*Après que nous avons obtenu le pivot, nous l'échangeons avec l'ouvrage situé à l'extrême gauche du groupe D. Dans notre exemple, nous arrivons à la disposition des ouvrages illustrée à la page 51.*

Nous comparons une fois chaque ouvrage avec le pivot. Chaque livre dont l'auteur vient avant celui du pivot ou est identique déclenche un échange. Pour partitionner $n$ livres, nous réalisons donc $n - 1$ comparaisons (puisque nous ne comparons pas le pivot avec lui-même) et au plus $n$ échanges. Vous noterez que, contrairement à la fusion, nous pouvons partitionner les ouvrages sans les retirer de l'étagère. Autrement dit, le partitionnement se fait sur place.

Pour traduire le partitionnement des ouvrages en partitionnement d'un sous-tableau $A[p..r]$, nous commençons par sélectionner $A[r]$ (l'élément à l'extrême droite) pour faire office de pivot. Ensuite, nous parcourons le sous-tableau de gauche à droite, en comparant chaque élément avec le pivot. Nous gérons des indices $q$ et $u$ dans le sous-tableau afin qu'ils le divisent de la manière suivante :

- Le sous-tableau $A[p..q-1]$ correspond au groupe G : chaque élément est inférieur ou égal au pivot.
- Le sous-tableau $A[q..u-1]$ correspond au groupe D : chaque élément est supérieur au pivot.
- Le sous-tableau $A[u..r-1]$ correspond au groupe I : nous ne savons pas encore comment les éléments se comparent au pivot.
- L'élément $A[r]$ correspond au groupe P : il contient le pivot.

Copyright © 2013 Dunod.

Ces divisions sont en réalité des invariants de boucle (mais nous ne les prouverons pas).

Lors de chaque étape, nous comparons $A[u]$, l'élément le plus à gauche du groupe I, avec le pivot. Si $A[u]$ lui est supérieur, nous incrémentons $u$ pour déplacer vers la droite la ligne de séparation des groupes D et I. Si, à la place, $A[u]$ est inférieur ou égal au pivot, alors nous échangeons les éléments $A[q]$ (l'élément à l'extrême gauche du groupe D) et $A[u]$, puis nous incrémentons $q$ et $u$ de façon à déplacer vers la droite les lignes de séparation entre les groupes G et D et les groupes D et I. Voici la procédure PARTITIONNER :

*Procédure* PARTITIONNER$(A, p, r)$

*Entrées :* Identiques à celles de TRI-FUSION.

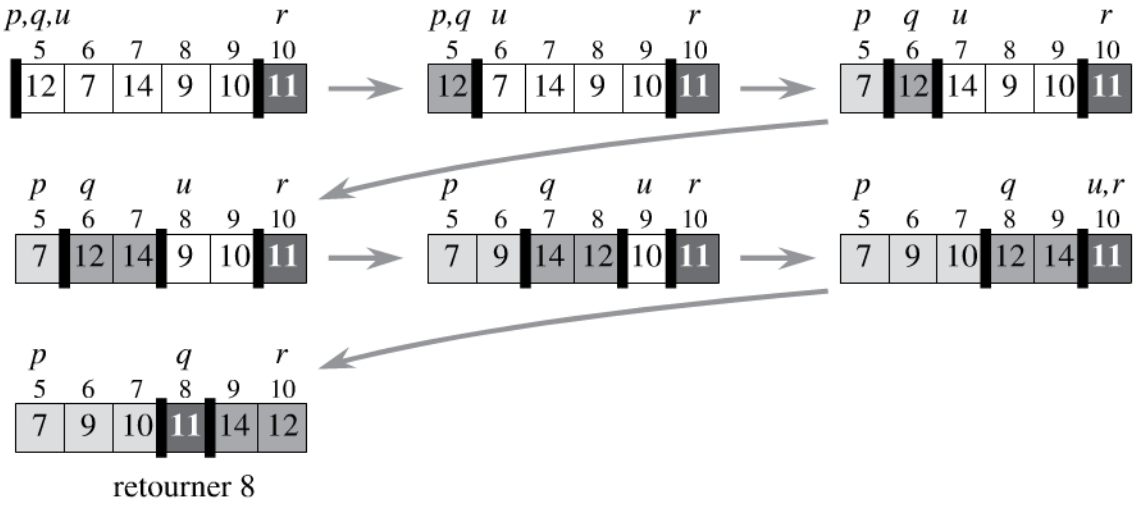
*Résultat :* Les éléments de $A[p..r]$ sont réorganisés de sorte que chaque élément de $A[p..q-1]$ est inférieur ou égal à $A[q]$ et que chaque élément de $A[q+1..r]$ est supérieur à $q$. L'indice $q$ est retourné à l'appelant.

1. Fixer $q$ à $p$.
2. Pour $u = p$ à $r-1$ :
   A. Si $A[u] \leq A[r]$, alors échanger $A[q]$ et $A[u]$, puis incrémenter $q$.
3. Échanger $A[q]$ et $A[r]$, puis retourner $q$.

Puisque les indices $q$ et $u$ sont initialement fixés à $p$, les groupes G ($A[p..q-1]$) et D ($A[q..u-1]$) sont identiques, et le groupe I ($A[u..r-1]$) contient tous les éléments à l'exception du pivot. Dans certains cas, par exemple lorsque $A[p] \leq A[r]$, un élément peut être échangé avec lui-même, ce qui ne modifie pas le tableau. L'étape 3 se termine en échangeant l'élément du pivot avec celui situé à l'extrême gauche du groupe D, déplaçant donc le pivot à l'emplacement adéquat dans le tableau partitionné, et retourne le nouvel indice $q$ du pivot.

Voici le déroulement étape par étape de la procédure PARTITIONNER sur le sous-tableau $A[5..10]$ créé par le premier partitionnement de l'exemple de tri rapide de la page 52. Le groupe I est représenté en blanc, le groupe L en gris clair, le groupe D en gris plus foncé et le groupe P du pivot en gris très foncé. La première partie de la figure montre le tableau et les indices initiaux. Les cinq parties suivantes montrent ces mêmes informations après chaque itération de la boucle de l'étape 2 (y compris l'incrémentation de l'indice $u$ à la fin de chaque tour de boucle). La dernière partie montre le tableau final partitionné.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.




Pendant le partitionnement des ouvrages, nous comparons une fois chaque élément avec le pivot et effectuons au plus un échange de chacun d'eux avec le pivot. Puisque chaque comparaison et chaque échange se font en un temps constant, le temps total de PARTITIONNER sur un sous-tableau de $n$ éléments est $\Theta(n)$.

Quel est donc le temps d'exécution de la procédure TRI-RAPIDE ? Comme dans le cas du tri par fusion, nous disons que le tri d'un sous-tableau de $n$ éléments se fait un temps $T(n)$, c'est-à-dire une fonction qui croît avec $n$. La division, mise en place par la procédure PARTITIONNER, demande un temps $\Theta(n)$. Cependant, la durée de TRI-RAPIDE dépend de l'uniformité du partitionnement.

Dans le cas le plus défavorable, les tailles des partitions sont réellement déséquilibrées. Si chaque élément est inférieur au pivot, alors PARTITIONNER laisse le pivot en $A[r]$ et retourne l'indice $r$ à TRI-RAPIDE, que celle-ci mémorise dans la variable $q$. La partition $A[q+1..r]$ est donc vide et la partition $A[p..q-1]$ ne comprend qu'un seul élément de moins que $A[p..r]$. L'appel récursif sur le sous-tableau vide prend un temps $\Theta(1)$ (la durée nécessaire à effectuer l'appel et à déterminer que le sous-tableau est vide, à l'étape 1). Nous pouvons intégrer ce $\Theta(1)$ dans le temps $\Theta(n)$ du partitionnement. En revanche, si $A[p..r]$ contient $n$ éléments, alors $A[p..q-1]$ possède $n-1$ éléments et l'appel récursif pour $A[p..q-1]$ demande un temps $T(n-1)$. Nous obtenons ainsi l'équation de récurrence suivante :

$$T(n) = T(n-1) + \Theta(n)$$

Nous ne sommes pas en mesure de résoudre cette récurrence en utilisant la méthode générale, mais sa solution est que $T(n)$ est $\Theta(n^2)$. Le résultat n'est pas meilleur que pour le tri par sélection ! Dans quel cas pouvons-nous avoir une répartition si peu uniforme ? Si chaque pivot est supérieur à tous les autres éléments, alors le tableau est initialement trié. Nous rencontrons également cette situation lorsque le tableau est trié en ordre inverse.

Copyright © 2013 Dunod.

D'autre part, si nous avons à chaque fois une répartition égale, chacun des sous-tableaux doit avoir au moins $n / 2$ éléments. L'équation de récurrence est alors identique à celle du tri par fusion donnée à la page 50 :

$$T(n) = 2T(n / 2) + \Theta(n)$$

La solution est identique. : $T(n)$ est $\Theta(n \lg n)$. Bien entendu, pour obtenir à chaque fois une telle répartition, il faudrait être vraiment chanceux ou avoir un tableau d'entrée peu réaliste.

Le cas normal se situe quelque part entre ces deux extrêmes. L'analyse technique est laborieuse et nous n'allons pas la réaliser, mais, si les éléments du tableau d'entrée sont dans un ordre aléatoire, alors, en moyenne, nous obtenons des partitions suffisamment égales pour que le temps d'exécution de TRI-RAPIDE soit $\Theta(n \lg n)$.

Faisons à présent dans la paranoïa et supposons que notre pire ennemi nous ait fourni un tableau à trier en sachant que nous prenons toujours le dernier élément de chaque sous-tableaux comme pivot et qu'il ait organisé les éléments de sorte que nous rencontrions toujours la plus mauvaise répartition. Comment pouvons-nous déjouer sa tactique ? Nous pouvons commencer par vérifier si le tableau n'est pas déjà trié (dans l'ordre ou dans l'ordre inverse) et procéder alors de manière particulière. Mais notre ennemi aurait pu préparer le tableau afin que les divisions soient toujours mauvaises, mais pas nécessairement les plus mauvaises possibles. Nous ne pouvons pas vérifier chaque cas imaginable.

Heureusement, il existe une solution beaucoup plus simple : il suffit de ne pas toujours choisir comme pivot le dernier élément. Cependant, notre procédure PARTITIONNER ne fonctionne plus car les groupes ne sont pas là où ils devraient être. Mais ce n'est pas un problème : avant d'exécuter PARTITIONNER, nous échangeons $A[r]$ avec un élément choisi aléatoirement dans $A[p..r]$. Nous avons à présent sélectionné notre pivot de façon aléatoire et nous pouvons exécuter la procédure PARTITIONNER.

En réalité, avec un petit effort supplémentaire, nous pouvons encore améliorer nos chances d'obtenir une séparation uniforme. Au lieu de choisir aléatoirement un élément dans $A[p..r]$, nous prenons trois éléments au hasard et échangeons celui du milieu avec $A[r]$. L'élément du milieu est celui dont la valeur se trouve entre celles des deux autres (si deux éléments ou plus choisis aléatoirement sont égaux, nous faisons une sélection arbitraire). À nouveau, je ne vais pas vous imposer l'analyse, mais il faudrait être vraiment malchanceux dans le choix des éléments aléatoires pour que TRI-RAPIDE prenne un temps supérieur à $\Theta(n \lg n)$. Par ailleurs, à moins que notre ennemi ait accès à notre générateur de nombres aléatoires, il n'aura aucun contrôle sur l'uniformité du partitionnement.

À combien d'échanges d'éléments procédons-nous dans TRI-RAPIDE ? Cela varie selon que nous tenons compte ou non des échanges d'un élément

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

à la même place. Nous pouvons vérifier si un élément se trouve dans ce cas et éviter l'échange. Par conséquent, considérons qu'il y a échange uniquement lorsqu'un élément se déplace réellement dans le tableau, c'est-à-dire lorsque $q \neq u$ à l'étape 2A ou lorsque $q \neq r$ à l'étape 3 de PARTITIONNER. Le cas le plus favorable pour la diminution du nombre d'échanges est également le pire pour le temps d'exécution asymptotique : lorsque le tableau est déjà trié, aucun échange n'a lieu. Le plus grand nombre d'échanges se produit lorsque $n$ est pair et que le tableau d'entrée a l'apparence suivante $n, n-2, n-4, \ldots, 4, 2, 1, 3, 5, \ldots, n-3, n-1$. Dans ce cas, $n^2 / 4$ échanges sont effectués et le temps d'exécution asymptotique et celui du cas le plus défavorable, $\Theta(n^2)$.

## RÉCAPITULATIF

Dans ce chapitre et le précédent, nous avons décrit quatre algorithmes de recherche et quatre algorithmes de tri. Nous résumons leurs caractéristiques dans deux tableaux. Puisque les trois algorithmes de recherche du Chapitre 2 n'étaient que des variantes sur le même thème, nous considérons que MEILLEURE-RECHERCHE-LINÉAIRE ou que RECHERCHE-LINÉAIRE-SENTINELLE est représentative de la recherche linéaire.

**Algorithmes de recherche**

| Algorithme | Temps d'exécution dans le cas le plus défavorable | Temps d'exécution dans le cas le plus favorable | Exige un tableau trié ? |
|---|---|---|---|
| Recherche linéaire | $\Theta(n)$ | $\Theta(1)$ | Non |
| Recherche dichotomique | $\Theta(\lg n)$ | $\Theta(1)$ | Oui |

Ces tableaux ne précisent pas le temps d'exécution dans le cas normal, car, à l'exception notable du tri rapide, il correspond au temps d'exécution dans le cas le plus défavorable. Comme nous l'avons vu, le temps d'exécution du tri rapide dans le cas normal, en supposant que le tableau soit initialement dans un ordre aléatoire, est uniquement $\Theta(n \lg n)$.

Copyright © 2013 Dunod.

Algorithmes de tri

| Algorithme | Temps d'exécution dans le cas le plus défavorable | Temps d'exécution dans le cas le plus favorable | Échanges dans le cas le plus défavorable | Opère sur place ? |
|---|---|---|---|---|
| Tri par sélection | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ | Oui |
| Tri par insertion | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ | Oui |
| Tri par fusion | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | Oui |
| Tri rapide | $\Theta(n^2)$ | $\Theta(n \lg n)$ | $\Theta(n^2)$ | Oui |

Dans la pratique, comment ces algorithmes se comportent-ils ? Je les ai mis en œuvre en C++ et les ai exécutés sur des tableaux d'entiers de quatre octets sur différentes machines : un MacBook Pro sous Mac OS 10.6.8, équipé d'un processeur Intel Core 2 Duo à 2,4 GHz et de 4 Go de RAM, et un PC Dell sous Linux version 2.6.22.14, équipé d'un processeur Intel Pentium 4 à 3,2 GHz et de 1 Go de RAM. Le code a été compilé avec g++ et une optimisation de niveau 3. Chaque algorithme a été appliqué à des tableaux allant jusqu'à 50 000 éléments et initialement triés en ordre inverse. J'ai effectué la moyenne des temps pour vingt exécutions de chaque algorithme sur chaque taille de tableau.

En partant d'un tableau trié dans l'ordre inverse, je me suis mis dans la situation du temps d'exécution asymptotique pour le cas le plus défavorable du tri par insertion et du tri rapide. C'est pourquoi j'ai exécuté deux versions du tri rapide : la version « normale », qui prend toujours comme pivot le dernier élément $A[r]$ du sous-tableau $A[p..r]$ en cours de partitionnement, et la version « aléatoire », qui échange un élément choisi aléatoirement dans $A[p..r]$ avec $A[r]$ avant de procéder au partitionnement (je n'ai pas mis en application la méthode de l'élément médian). La version normale du tri rapide est également connue comme **déterministe** car le hasard n'entre pas en ligne de compte ; tout est déjà prédéterminé dès le moment où le tableau d'entrée à trier est fourni.

La version aléatoire du tri rapide a été la meilleure sur les deux ordinateurs pour $n \geq 64$. Voici les rapports des temps d'exécution des autres algorithmes par rapport à celui du tri rapide aléatoire en fonction des différentes tailles du tableau d'entrée.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

MacBook Pro

| | *n* | | | | | | |
|---|---|---|---|---|---|---|---|
| **Algorithme** | **50** | **100** | **500** | **1 000** | **5 000** | **10 000** | **50 000** |
| Tri par sélection | 1,34 | 2,13, | 8,04 | 13,31 | 59,07 | 114,24 | 537,42 |
| Tri par insertion | 1,08 | 2,02 | 6,15 | 11,35 | 51,86 | 100,38 | 474,29 |
| Tri par fusion | 7,58 | 7,64 | 6,93 | 6,87 | 6,35 | 6,20 | 6,27 |
| Tri rapide déterministe | 1,02 | 1,63 | 6,09 | 11,51 | 52,02 | 100,57 | 475,34 |

PC Dell Pro

| | *n* | | | | | | |
|---|---|---|---|---|---|---|---|
| **Algorithme** | **50** | **100** | **500** | **1 000** | **5 000** | **10 000** | **50 000** |
| Tri par sélection | 0,76 | 1,60 | 5,46 | 12,23 | 52,03 | 100,79 | 496,94 |
| Tri par insertion | 1,01 | 1,66 | 7,68 | 13,90 | 68,34 | 136,20 | 626,44 |
| Tri par fusion | 3,21 | 3,38 | 3,57 | 3,33 | 3,36 | 3,37 | 3,15 |
| Tri rapide déterministe | 1,12 | 1,37 | 6,52 | 9,30 | 47,60 | 97,45 | 466,83 |

Le tri rapide aléatoire semble plutôt bon, mais nous pouvons faire mieux. Rappelez-vous que le tri par insertion fonctionne parfaitement lorsqu'aucun élément ne doit être déplacé très loin dans le tableau. Lorsque les tailles du sous-problème des algorithmes récursifs arrivent à une limite inférieure $k$, aucun élément n'a besoin d'être déplacé de plus de $k - 1$ positions. Au lieu de poursuivre les appels récursifs du tri rapide aléatoire après que les tailles du sous-problème sont devenues faibles, que se passerait-il si nous exécutions un tri par insertion habilement modifié pour opérer sur un sous-tableau plutôt que sur un tableau entier ? Avec une telle méthode hybride, nous pouvons aller plus vite que le tri rapide aléatoire. Sur mon MacBook Pro, j'ai pu déterminer qu'un sous-tableau de taille 22 représentait le point de basculement optimal, tandis qu'il était atteint avec une taille 17 sur le PC. Voici les rapports des temps d'exécution de l'algorithme hybride par rapport au tri rapide aléatoire sur les deux machines, pour les mêmes tailles de problème.

Copyright © 2013 Dunod.

| | *n* | | | | | | |
|---|---|---|---|---|---|---|---|
| **Machine** | **50** | **100** | **500** | **1 000** | **5 000** | **10 000** | **50 000** |
| MacBook Pro | 0,55 | 0,56 | 0,60 | 0,60 | 0,62 | 0,63 | 0,66 |
| PC | 0,53 | 0,58 | 0,60 | 0,58 | 0,60 | 0,64 | 0,64 |

Est-il possible de battre un temps $\Theta(n \lg n)$ dans le cas du tri ? Cela dépend. Nous verrons au Chapitre 4 que, si la seule manière de déterminer l'emplacement des éléments consiste à les comparer, en effectuant des opérations fondées sur les résultats de la comparaison, alors non nous ne pouvons pas faire mieux qu'un temps $\Theta(n \lg n)$. En revanche, si nous avons sur les éléments des informations que nous pouvons exploiter, alors oui nous pouvons être plus rapides.

## Lectures complémentaires

Le CLRS [CLRS09] détaille le tri par insertion, le tri par fusion et les versions déterministes et aléatoires du tri rapide. Néanmoins, le père de tous les ouvrages sur le tri et la recherche reste le volume 3 de *The Art of Computer Programming* par Knuth [Knu98b]. Notre avis donné au Chapitre 1 est toujours de mise : le TAOCP est d'une lecture ardue.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# 4 Minorant pour le tri à battre

Au chapitre précédent, nous avons étudié quatre algorithmes de tri d'un tableau de $n$ éléments. Deux d'entre eux, le tri par sélection et le tri par insertion, présentent un temps d'exécution de $\Theta(n^2)$ dans le cas le plus défavorable, ce qui n'est pas très bon. Un autre, le tri rapide, affiche également un temps d'exécution de $\Theta(n^2)$ dans le cas le plus défavorable, mais uniquement de $\Theta(n \lg n)$ dans le cas normal. Le tri par fusion a un temps $\Theta(n \lg n)$ dans tous les cas. Dans la pratique, le tri rapide est le meilleur des quatre, mais si nous pouvons absolument éviter le cas le plus défavorable, le tri par fusion doit être sélectionné.

$\Theta(n \lg n)$ est-il aussi bon que cela ? Est-il possible d'imaginer un algorithme de tri dont le temps d'exécution est meilleur que $\Theta(n \lg n)$ dans le cas le plus défavorable ? La réponse dépend des règles du jeu : quelle utilisation des clés de tri l'algorithme de tri est-il autorisé à faire lorsqu'il détermine l'ordre trié ?

Dans ce chapitre, nous verrons toute d'abord que certaines règles ne permettent pas d'obtenir mieux que $\Theta(n \lg n)$. Ensuite, nous présenterons des algorithmes de tri, par comptage et par base, qui tordent les règles afin d'arriver à un temps de seulement $\Theta(n)$.

## Règles pour le tri

Si nous examinons l'utilisation des clés de tri dans les quatre algorithmes du chapitre précédent, nous constatons qu'ils déterminent l'ordre trié en fonction du résultat de la comparaison d'un couple de clés. Toutes les décisions prises sont de la forme « si la clé de tri de cet élément est inférieure à celle de cet autre élément, alors faire ceci, sinon faire cela ou ne rien faire du tout ». Vous pourriez penser qu'un algorithme de tri n'a pas d'autre forme de décision.

Nous allons voir que ce n'est pas le cas et, pour cela, nous prenons un cas réellement simple. Supposons que nous sachions deux choses sur les éléments à trier : chaque clé de tri vaut 1 ou 2, et les éléments sont constitués

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

uniquement de clés de tri (aucunes données satellites). Dans cette configuration simple, nous pouvons trier $n$ éléments en un temps de seulement $\Theta(n)$, battant ainsi les algorithmes en $\Theta(n \lg n)$ du chapitre précédent. Comment est-ce possible ? Tout d'abord, nous parcourons chaque élément et comptons le nombre d'entre eux qui sont des 1 ; supposons que $k$ ait la valeur 1. Ensuite, nous parcourons le tableau en plaçant la valeur 1 dans les $k$ premières positions, puis la valeur 2 dans les $n - k$ emplacements restants. Voici la procédure correspondante :

*Procédure* TRI-TRÈS-SIMPLE $(A, n)$
*Entrées :*
- $A$ : un tableau dont chaque élément vaut 1 ou 2.
- $n$ : le nombre d'éléments dans $A$.

*Résultat :* les éléments de $A$ sont triés par ordre croissant.

1. Fixer $k$ à 0.
2. Pour $i = 1$ à $n$ :
   A. Si $A[i] = 1$, alors incrémenter $k$.
3. Pour $i = 1$ à $k$ :
   A. Fixer $A[i]$ à 1.
3. Pour $i = k + 1$ à $n$ :
   A. Fixer $A[i]$ à 2.

Les étapes 1 et 2 comptent le nombre de 1, en incrémentant $k$ pour chaque élément $A[i]$ égal à 1. L'étape 3 remplit $A[1..k]$ avec des 1, tandis que l'étape 4 remplit les emplacements restants, $A[k + 1.. n]$, avec des 2. Il est assez facile de conclure que cette procédure a un temps d'exécution $\Theta(n)$ : la première boucle réalise $n$ itérations, les deux dernières effectuent ensemble $n$ itérations, qui se font chacune en un temps constant.

Notez que TRI-TRÈS-SIMPLE ne compare jamais deux éléments du tableau *l'un avec l'autre*, mais chaque élément à la valeur 1. Dans ce cas parfaitement restreint, nous pouvons effectuer un tri sans comparer des clés de tri.

## MINORANT POUR LE TRI PAR COMPARAISON

Maintenant que vous savez comment les règles du jeu peuvent varier, voyons un minorant de la rapidité du tri.

Un **tri par comparaison** correspond à un algorithme de tri qui détermine l'ordre en comparant des couples d'éléments. Les quatre algorithmes de tri du chapitre précédent sont des tris par comparaison, contrairement à TRI-TRÈS-SIMPLE.

Copyright © 2013 Dunod.

Voici le minorant :

> Dans le cas le plus défavorable, un algorithme de tri par comparaison sur $n$ éléments nécessite $\Omega(n \lg n)$ comparaisons entre des couples d'éléments.

Rappelons que la notation $\Omega$ donne une borne inférieure. Autrement dit, nous disons que pour $n$ suffisamment grand, n'importe quel algorithme de tri par comparaison implique au moins $cn \lg n$ comparaisons dans le cas le plus défavorable, pour une constante $c$. Puisque chaque comparaison se fait au moins en un temps constant, nous avons un minorant $\Omega(n \lg n)$ sur le temps de tri par comparaison de $n$ éléments.

Ce minorant présente deux points qu'il est important de comprendre. Tout d'abord, il s'applique uniquement au cas le plus défavorable. Nous pouvons toujours écrire un algorithme de tri qui s'exécute en un temps linéaire dans le cas le plus favorable : il suffit de déclarer que le cas le plus favorable correspond à un tableau déjà trié et de vérifier simplement que chaque élément (à l'exception du dernier) est inférieur ou égal à son successeur dans le tableau. Cela est facile à réaliser en un temps $\Theta(n)$ et si nous déterminons que chaque élément est inférieur ou égal à son successeur, alors l'affaire est réglée. Toutefois, dans le *cas le plus défavorable*, il faut $\Omega(n \lg n)$ comparaisons. Il s'agit d'un minorant **existentiel** car il indique qu'il existe une entrée qui nécessite $\Omega(n \lg n)$ comparaisons. Tout autre type de borne inférieure est un minorant **universel** qui s'applique à toutes les entrées. Dans le cas du tri, le seul minorant universel que nous ayons, est $\Omega(n)$ car nous devons examiner chaque élément au moins une fois Vous aurez peut-être remarqué que la phrase précédente ne précise pas $\Omega(n)$ *quoi*. Sous-entend-elle $\Omega(n)$ comparaisons ou un temps $\Omega(n)$ ? Il s'agit d'un temps $\Omega(n)$ car nous devons examiner chaque élément même si nous ne comparons pas des couples.

Le deuxième point important est véritablement remarquable : le minorant ne dépend pas d'un algorithme particulier, tant qu'il s'agit d'un tri par comparaison. Il s'applique à *tous* les algorithmes de tri par comparaison, qu'ils soient simples ou complexes, qu'ils aient déjà été inventés ou qu'ils restent à découvrir. Il s'applique même à ceux qui ne seront jamais découverts par l'humanité !

## Dépasser le minorant avec le tri par dénombrement

Nous avons déjà vu comment dépasser le minorant dans une configuration très restreinte : les clés de tri ont uniquement deux valeurs possibles et chaque élément est constitué uniquement d'une clé de tri, sans données satellites.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans ce cadre borné nous pouvons trier $n$ éléments en un temps $\Theta(n)$ sans comparer des couples d'éléments.

Nous pouvons généraliser la méthode de TRI-TRÈS-SIMPLE de façon à prendre en charge $m$ valeurs différentes pour les clés de tri, tant qu'il s'agit d'entiers dans une plage de $m$ entiers consécutifs, disons de 0 à $m - 1$, et autoriser les données satellites dans les éléments.

Voici le principe. Supposons que nous sachions que les clés de tri soient des entiers dans la plage 0 à $m - 1$ et que précisément trois éléments ont des clés de tri égales à 5 et que six ont des clés de tri inférieures à 5 (autrement dit dans la plage 0 à 4). Dans ces conditions, nous savons que, dans le tableau trié, les éléments qui ont des clés de tri égales à 5 occuperont les emplacements 7, 8 et 9. En généralisant le raisonnement, si nous savons que $k$ éléments ont des clés de tri égales à $x$ et que $l$ éléments ont des clés de tri inférieures à $x$, alors nous savons que les éléments dont les clés de tri sont égales à $x$ occuperont les emplacements $l + 1$ à $l + k$ dans le tableau trié. Par conséquent, nous allons calculer, pour chaque valeur de clé de tri possible, le nombre d'éléments dont les clés de tri sont inférieures à cette valeur et le nombre d'éléments dont les clés de tri sont égales à cette valeur.

Pour déterminer le nombre d'éléments dont les clés de tri sont inférieures à chaque valeur de clé de tri possible, nous calculons tout d'abord combien d'éléments ont des clés de tri égales à cette valeur :

*Procédure* DÉNOMBRER-CLÉS-ÉGALES($A$,$n$,$m$)
*Entrées :*
- $A$ : un tableau d'entiers dans la plage 0 à $m - 1$.
- $n$ : le nombre d'éléments dans $A$.
- $m$ : définit la plage des valeurs dans $A$.

*Sortie :* Un tableau *égal*[0..$m - 1$] tel que *égal*[$j$] contient le nombre d'éléments de $A$ égaux à $j$, pour $j = 0,1,2,\ldots,m - 1$.

1. Créer nouveau tableau *égal*[0..$m - 1$].
2. Fixer toutes les valeurs de *égal* à 0.
3. Pour $i = 1$ à $n$ :
   A. Fixer *clé* à $A[i]$.
   B. Incrémenter *égal*[*clé*].
4. Retourner le tableau *égal*.

Vous remarquerez que DÉNOMBRER-CLÉS-ÉGALES ne compare jamais des clés de tri les unes avec les autres. La procédure utilise les clés de tri uniquement comme indice dans le tableau *égal*. Puisque la première boucle

Copyright © 2013 Dunod.

(implicite dans l'étape 2) effectue $m$ itérations, que la seconde boucle (étape 3) effectue $n$ itérations et que chaque itération de chaque boucle se fait en un temps constant, DÉNOMBRER-CLÉS-ÉGALES prend un temps $\Theta(m + n)$. Si $m$ est une constante, alors le temps d'exécution de DÉNOMBRER-CLÉS-ÉGALES est $\Theta(n)$.

Nous pouvons à présent nous servir du tableau *égal* pour effectuer un total cumulé de façon à déterminer le nombre d'éléments qui ont des clés de tri inférieures à chaque valeur :

*Procédure* DÉNOMBRER-CLÉS-INFÉRIEURES(*égal*,*m*)

*Entrées :*

- *égal* : le tableau retourné par la procédure DÉNOMBRER-CLÉS-ÉGALES.
- $m$ : définit les plages des indices de *égal* : 0 à $m - 1$.

*Sortie :* Un tableau *inférieur*[0..$m - 1$] tel que pour $j = 0,1,2,\ldots,m - 1$, *inférieur*[$j$] contient la somme *égal*[0] + *égal*[1] + ... + *égal*[$j - 1$].

1. Créer nouveau tableau *inférieur*[0..$m - 1$].
2. Fixer *inférieur*[0] à 0.
3. Pour $j = 1$ à $m - 1$ :
   A. Fixer *inférieur*[$j$] à *inférieur*[$j - 1$] + *égal*[$j - 1$].
4. Retourner le tableau *inférieur*.

En supposant que *égal*[$j$] donne le nombre exact de clés de tri qui sont égales à $j$, pour $j = 0,1,\ldots,m - 1$, nous pouvons utiliser l'invariant de boucle suivant pour montrer que, lorsque DÉNOMBRER-CLÉS-INFÉRIEURES retourne, *inférieur*[$j$] indique le nombre de clés de tri qui sont inférieures à $j$ :

> Au début de chaque itération de la boucle à l'étape 3, *inférieur*[$j - 1$] est égal au nombre de clés de tri inférieures à $j - 1$.

Je vous laisse le soin de développer l'argumentation. Vous pouvez facilement constater que la procédure DÉNOMBRER-CLÉS-INFÉRIEURES s'exécute en un temps $\Theta(m)$. Par ailleurs, il est certain qu'elle ne compare pas les clés de tri entre elles.

Prenons un exemple. Supposons que $m = 7$, que toutes les clés de tri sont donc des entiers dans la plage 0 à 6 et que nous ayons le tableau $A$ suivant avec $n = 10$ éléments : $A = \langle 4,1,5,0,1,6,5,1,5,3\rangle$. Alors *égal*$\langle 1,3,0,1,1,3,1\rangle$ et *inférieur* $= \langle 0,1,4,4,5,6,9\rangle$. Puisque *inférieur*[5] = 6 et que *égal*[5] = 3 (rappelons que les indices des tableaux *inférieur* et *égal* commencent non pas à 1 mais à 0), lorsque le tri est terminé, les emplacements 1 à 6 doivent contenir des valeurs de clés inférieures à 5, et les emplacements 7, 8 et 9, la valeur de clé 5.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

À partir du tableau *inférieur*, nous pouvons créer un tableau trié, mais sans pouvoir opérer sur place :

*Procédure* RÉORGANISER($A$,*inférieur*,$n$,$m$)
*Entrées :*

- $A$ : un tableau d'entiers dans la plage 0 à $m - 1$.
- *inférieur* : le tableau retourné par DÉNOMBRER-CLÉS-INFÉRIEURES.
- $n$ : le nombre d'éléments dans $A$.
- $m$ : définit la plage des valeurs dans $A$.

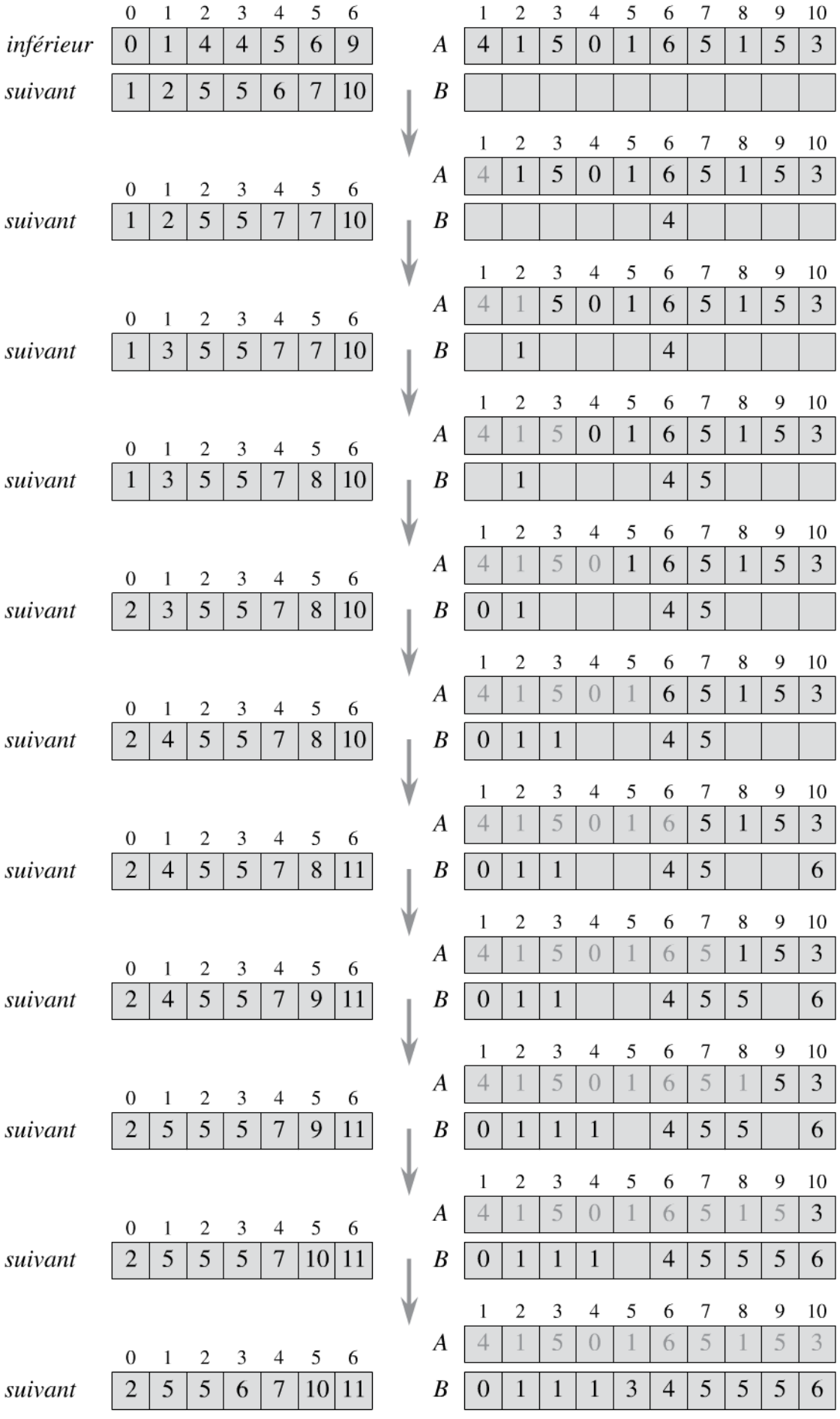
*Sortie :* Un tableau $B$ qui contient les éléments de $A$ triés.

1. Créer les tableaux $B[1..n]$ et *suivant*$[0..m - 1]$.
2. Pour $j = 0$ à $m - 1$ :
   A. Fixer *suivant*$[j]$ à *inférieur*$[j]$+1.
3. Pour $i = 1$ à $n$ :
   A. Fixer *clé* à $A[i]$.
   B. Fixer *indice* à *suivant*[*clé*].
   C. Fixer $B$[*indice*] à $A[i]$.
   D. Incrémenter *suivant*[*clé*].
4. Retourner le tableau $B$.

La figure suivante (page 69) illustre le déplacement des éléments dans les tableaux $A$ et $B$ par la procédure RÉORGANISER de façon à les trier dans $B$. La partie supérieure montre les tableaux *inférieur*, *suivant*, $A$ et $B$ avant la première itération de la boucle de l'étape 3. Chaque partie suivante montre les tableaux *suivant*, $A$ et $B$ après chaque itération. Les éléments de $A$ recopiés dans $B$ sont indiqués en gris clair.

L'idée est que, pendant le parcours du tableau $A$ du début à la fin, *suivant*$[j]$ donne l'indice dans le tableau $B$ où placer l'élément suivant de $A$ dont la clé est $j$. Rappelons que si $l$ éléments ont des clés de tri inférieures à $x$, alors les $k$ éléments dont les clés de tri sont égales à $x$ doivent occuper les emplacements $l + 1$ à $l + k$. La boucle de l'étape 2 configure le tableau *suivant* afin que, au début, *suivant*$[j] = l + 1$, où $l$ = *inférieur*$[j]$. La boucle de l'étape 3 parcourt le tableau $A$ du début à la fin. Pour chaque élément $A[i]$, l'étape 3A place $A[i]$ dans *clé*, l'étape 3B calcule *indice* afin qu'il corresponde à l'indice dans $B$ où doit aller $A[i]$, et l'étape 3C déplace $A[i]$ dans cet emplacement de $B$. Puisque l'élément suivant du tableau $A$ qui possède la même clé de tri que $A[i]$ (s'il existe) doit aller à l'emplacement suivant dans $B$, l'étape 3D incrémente *suivant*[*clé*].

Copyright © 2013 Dunod.




Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Quel est le temps d'exécution de RÉORGANISER ? La boucle de l'étape 2 demande un temps $\Theta(m)$, tandis que celle de l'étape 3 prend un temps $\Theta(n)$. À l'instar de DÉNOMBRER-CLÉS-ÉGALES, RÉORGANISER s'exécute donc en un temps $\Theta(m + n)$, ou $\Theta(n)$ si $m$ est une constante.

Nous pouvons à présent réunir les trois procédures pour créer le tri par dénombrement :

*Procédure* TRI-DÉNOMBREMENT($A$,$n$,$m$)

*Entrées* :

- $A$ : un tableau d'entiers dans la plage 0 à $m - 1$.
- $n$ : le nombre d'éléments dans $A$.
- $m$ : définit la plage des valeurs dans $A$.

*Sortie :* Un tableau $B$ qui contient les éléments de $A$ triés.

1. Appeler DÉNOMBRER-CLÉS-ÉGALES($A$,$n$,$m$) et affecter son résultat à *égal*.
2. Appeler DÉNOMBRER-CLÉS-INFÉRIEURES(*égal*,$m$) et affecter son résultat à *inférieur*.
3. Appeler RÉORGANISER($A$,*inférieur*,$n$,$m$) et affecter son résultat à $B$.
4. Retourner le tableau $B$.

À partir des temps d'exécution de DÉNOMBRER-CLÉS-ÉGALES ($\Theta(m + n)$), de DÉNOMBRER-CLÉS- INFÉRIEURES ($\Theta(m)$) et de RÉORGANISER ($\Theta(m + n)$), nous pouvons en conclure que TRI-DÉNOMBREMENT a un temps d'exécution $\Theta(m + n)$, ou $\Theta(n)$ si $m$ est une constante. Le tri par dénombrement dépasse donc le minorant $\Omega(n \lg n)$ du tri par comparaison car il ne compare jamais des clés de tri les unes avec les autres. À la place, il s'en sert comme des indices dans des tableaux, ce qui est possible car elles sont de petits entiers. Si les clés de tri étaient des nombres réels, avec des parties fractionnaires, ou des chaînes de caractères, le tri par dénombrement ne pourrait pas s'appliquer.

Vous aurez peut-être remarqué que la procédure suppose que les éléments contiennent uniquement des clés de tri, sans données satellites. Pourtant, j'avais promis que, contrairement à TRI-TRÈS-SIMPLE, la procédure TRI-DÉNOMBREMENT acceptait de telles données. C'est effectivement le cas, à condition de modifier l'étape 3C de RÉORGANISER de manière à copier non pas uniquement la clé de tri mais l'intégralité de l'élément.

Par ailleurs, vous pourriez avoir noté que les procédures proposées sont plutôt inefficaces dans leur utilisation des tableaux. Il est possible de réunir les tableaux *égal*, *inférieur* et *suivant* en un seul, mais je vous laisse ce travail en exercice.

Copyright © 2013 Dunod.

À plusieurs reprises, j'ai indiqué que le temps d'exécution est $\Theta(n)$ si $m$ est une constante. Quand est-ce le cas ? Un exemple serait le tri des résultats aux examens en fonction des notes. Les notes vont de 0 à 100, mais le nombre d'étudiants varie. Il est possible d'employer le tri par dénombrement pour trier les examens de $n$ étudiants en un temps $\Theta(n)$ car $m = 101$ (la plage va de 0 à $m - 1$) est une constante.

Cependant, dans la pratique, le tri par dénombrement se révèle utile au sein d'un autre algorithme de tri, appelé tri par base, ou tri radix. Outre le fait de s'exécuter en un temps linéaire lorsque $m$ est une constante, le tri par dénombrement possède une autre propriété importante : il est **stable**. Dans un tri stable, les éléments qui possèdent la même clé de tri apparaissent dans le tableau de sortie dans le même ordre que dans le tableau d'entrée. Autrement dit, un tri stable départage deux éléments dont les clés de tri sont égales en plaçant en premier dans le tableau de sortie celui qui apparaît en premier dans le tableau d'entrée. Vous pouvez constater que le tri par dénombrement est stable en examinant la boucle de l'étape 3 de la procédure RÉORGANISER. Si deux éléments ont la même clé de tri, disons *clé*, alors la procédure augmente *suivant*[*clé*] immédiatement après avoir déplacé dans $B$ l'élément qui vient avant dans $A$ ; de cette manière, lors du déplacement de l'élément qui vient ensuite dans $A$, cet élément apparaît après dans $B$.

## TRI PAR BASE

Supposons que nous ayons à trier des chaînes de caractères de longueur fixe. Imaginons par exemple que je sois en train d'écrire ce paragraphe dans un avion et que, lorsque j'ai fait la réservation du vol, j'ai reçu le code de confirmation XI7FS6. Tous ces codes sont fournis sous forme de chaînes de six caractères, constituées de lettres et de chiffres. Puisque chaque caractère peut avoir 36 valeurs (26 lettres plus 10 chiffres), il y a $36^6 = 2\,176\,782\,336$ codes de confirmation possibles. Il s'agit certes d'une constante, mais plutôt grande, et la compagnie préférera sans doute ne pas utiliser un tri par dénombrement pour trier les codes de confirmation.

Pour être concrets, supposons que nous convertissions chacun des 36 caractères en un code numérique allant de 0 à 35. Le code d'un chiffre est le chiffre lui-même (celui du chiffre 5 est donc 5) et les codes des lettres débutent à 10 pour le A et vont jusqu'à 35 pour le Z.

Simplifions un peu les choses et supposons que chaque code de confirmation ne comprenne que deux caractères (ne vous inquiétez pas, nous reviendrons aux six caractères). Bien que nous puissions exécuter le tri par dénombrement avec $m = 36^2 = 1\,296$, nous allons l'exécuter deux fois avec $m = 36$. La première utilisera le caractère à l'extrême droite comme clé de tri. Nous prendrons ensuite le résultat de cette première exécution et procéderons à une deuxième exécution en utilisant cette fois-ci le caractère

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

à l'extrême gauche comme clé de tri. Nous choisissons le tri par dénombrement car il fonctionne parfaitement lorsque $m$ est relativement petit et parce qu'il est stable.

Supposons que les codes de confirmation sur deux caractères soient <F6,E5,R6,X6,X2,T5,F2,T3>. Au terme de l'exécution du tri par dénombrement sur le caractère de droite, nous obtenons le tableau trié <X2,F2,T3,E5,T5,F6,R6,X6>. Puisque le tri par dénombrement est stable et puisque X2 vient avant F2 dans le tableau d'origine, X2 reste avant F2 après le tri sur le caractère de droite. Nous trions à présent le résultat sur le caractère de gauche, à nouveau en utilisant un tri par dénombrement. Nous obtenons le résultat souhaité, c'est-à-dire <E5,F2,F6,R6,T3,T5,X2,X6>.

Quel aurait été le résultat d'un tri initial sur le caractère de gauche ? Après le premier tri, nous aurions obtenu le tableau <E5,F6,F2,R6,T5,T3,X6,X2>, puis le tableau <F2,X2,T3,E5,T5,F6,R6,X6> après le deuxième tri sur le caractère de droite. Le résultat n'est donc pas correct.

Pourquoi le fait d'opérer de droite à gauche produit-il un résultat juste ? L'utilisation d'une méthode de tri stable est importante ; il peut s'agir d'un tri par dénombrement ou de n'importe quelle autre méthode de tri stable. Supposons que nous manipulions le $i$-ème emplacement de caractère et supposons que les $i - 1$ emplacements de caractères de droite soient triés. Prenons deux clés de tri quelconques. Si elles diffèrent dans le $i$-ème emplacement de caractère, alors peut importe ce qui se trouve dans les $i - 1$ emplacements de droite : l'algorithme de tri stable qui trie sur le $i$-ème emplacement les placera dans l'ordre correct. En revanche, si elles ont le même caractère au $i$-ème emplacement, alors celle qui vient en premier dans les $i - 1$ emplacements de caractères de droite doit être placée en premier et, en choisissant une méthode de tri stable, nous sommes certains que cela se passe ainsi.

Revenons aux codes de confirmation sur six caractères et voyons comment trier le tableau initial <X17FS6,PL4ZQ2,JI8FR9,KV7WS9, JL2ZV3,KI4WR2>. Numérotons les caractères de droite à gauche de 1 à 6. Voici le résultat de l'exécution d'un tri stable sur le $i$-ème caractère, en opérant de droite à gauche :

| $i$ | Ordre obtenu |
|---|---|
| 1 | <PL4ZQ2,KI4WR2,JL2ZV3,PY2ZR5,XI7FS6,XL8FQ6,JI8FR9,KV7WS9> |
| 2 | <PL4ZQ2,XL8FQ6,KI4WR2,PY2ZR5,JI8FR9,XI7FS6,KV7WS9,JL2ZV3> |
| 3 | <XL8FQ6,JI8FR9,XI7FS6,KI4WR2,KV7WS9,PL4ZQ2,PY2ZR5,JL2ZV3> |
| 4 | <PY2ZR5,JL2ZV3,KI4WR2,PL4ZQ2,XI7FS6,KV7WS9,XL8FQ6,JI8FR9> |
| 5 | <KI4WR2,XI7FS6,JI8FR9,JL2ZV3,PL4ZQ2,XL8FQ6,KV7WS9,PY2ZR5> |
| 6 | <JI8FR9,JL2ZV3,KI4WR2,KV7WS9,PL4ZQ2,PY2ZR5,XI7FS6,XL8FQ6> |

Copyright © 2013 Dunod.

Pour généraliser, dans l'algorithme de tri par base, nous considérons chaque clé de tri comme un nombre à $d$ chiffres, où chaque chiffre se trouve dans la plage 0 à $m - 1$. Nous exécutons un tri stable sur chaque chiffre, en allant de droite à gauche. Si nous employons un tri par dénombrement, alors le temps de tri sur un chiffre est $\Theta(m + n)$ et le temps de tri sur les $d$ chiffres est $\Theta(d(m + n))$. Si $m$ est une constante (comme 36 dans l'exemple des codes de confirmation), alors le temps du tri par base est $\Theta(dn)$. Si $d$ est également une constante (comme 6 pour les codes de confirmation), alors le temps du tri par base est uniquement $\Theta(n)$.

Lorsque le tri par base se fonde sur un tri par dénombrement sur chaque chiffre, il ne compare jamais des clés de tri. Il utilise les chiffres individuels comme indice dans des tableaux manipulés par le tri par dénombrement. C'est pourquoi le tri par base, à l'instar du tri par dénombrement, permet de dépasser le minorant $\Omega(n \lg n)$ du tri par comparaison.

## Lectures complémentaires

Le Chapitre 8 de CLRS [CLRS09] développe les sujets abordés dans ce chapitre.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# 5 GRAPHES ORIENTÉS ACYCLIQUES

À la page 1, vous avez appris que j'ai joué au hockey. C'était comme gardien de but, pendant plusieurs années. Cependant, mon niveau a fini par se dégrader, jusqu'au point où je n'osais même plus me regarder jouer. Chaque tir adverse parvenait à entrer dans la cage. Après une pause de plus de sept ans, je suis revenu entre les poteaux pour quelques matchs.

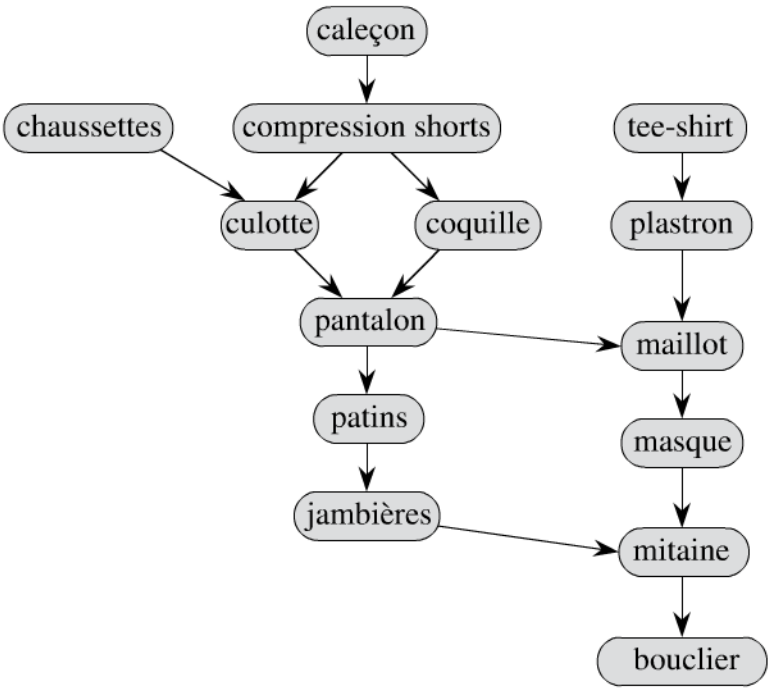
Ma principale préoccupation était non pas de savoir si je serais suffisamment bon (je n'avais aucun doute là-dessus), mais si je me rappellerais comment enfiler tout l'équipement du gardien de but.

En hockey sur glace, le gardien porte une tenue compliquée, qui pèse environ 20 kilos, et dont les différents éléments doivent être passés dans le bon ordre. Par exemple, puisque je suis droitier, ma main gauche est protégée par un gant de grande taille, la mitaine, qui permet d'attraper les palets. Lorsque le gant est en place, ma main gauche n'a plus aucune dextérité et il m'est impossible de passer un autre vêtement par-dessus.

Avant d'endosser le poste de gardien de but, j'avais pris le temps de tracer un diagramme qui indiquait dans quel ordre enfiler les différents éléments de la tenue. La figure ci-après le représente.

Une flèche qui va de l'article A à l'article B indique que A doit être mis avant B. Par exemple, je dois passer le plastron avant le maillot. Bien entendu, la contrainte « doit être passé avant » est **transitive** : si l'article A doit être passé avant l'article B et si article B doit être passé avant l'article C, alors l'article A doit être passé avant l'article C. Par conséquent, je dois mettre le plastron avant le maillot, le masque, la mitaine et le bouclier.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.



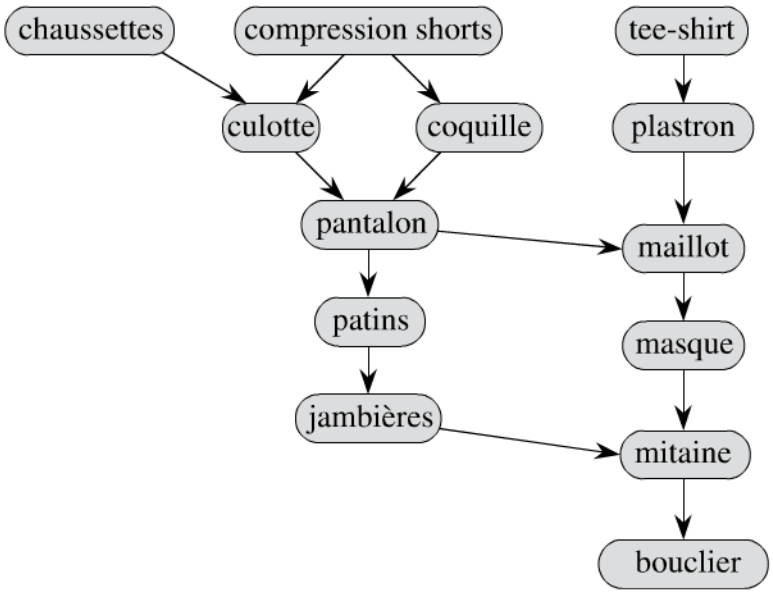

En revanche, pour certains articles, peu importe l'ordre dans lequel je les enfile. Par exemple, je peux mettre les chaussettes avant ou après le plastron.

Pour m'habiller, il m'a fallu déterminer un ordre. Dès que mon diagramme a été terminé, j'ai obtenu une liste qui comprenait tous les articles que je devais revêtir, dans un ordre unique qui respectait les différentes contraintes « doit être passé avant ». J'ai pu établir que plusieurs ordres étaient possibles ; le diagramme présenté correspond à l'un des trois donnés dans le tableau suivant.

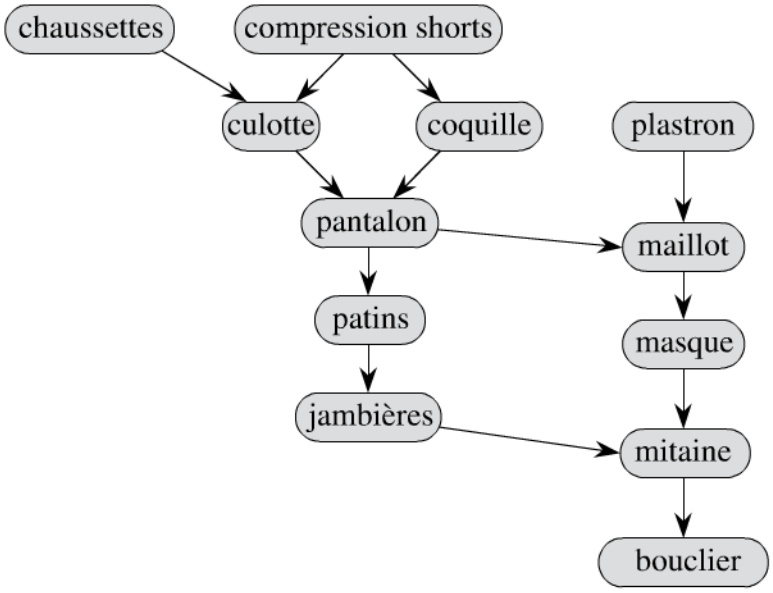
| Ordre n° 1 | Ordre n° 2 | Ordre n° 3 |
|---|---|---|
| caleçon | caleçon | chaussettes |
| compression shorts | tee-shirt | tee-shirt |
| coquille | gaine | caleçon |
| chaussettes | coquille | plastron |
| bouclier | plastron | gaine |
| pantalon | chaussettes | culotte |
| patins | culotte | coquille |
| jambières | pantalon | pantalon |
| tee-shirt | maillot | patins |
| plastron | masque | jambières |
| maillot | patins | maillot |
| masque | jambières | masque |
| mitaine | mitaine | mitaine |
| bouclier | bouclier | bouclier |

Copyright © 2013 Dunod.

Comment suis-je parvenu à ces différents ordres ? Voici la méthode que j'ai suivie pour obtenir l'ordre numéro 2. J'ai recherché un article pour lequel il n'y avait aucune flèche entrante, car un tel équipement n'a pas besoin d'être passé après un autre. J'ai décidé que le caleçon serait le premier article. Ensuite, après l'avoir (conceptuellement) enfilé, je l'ai retiré du diagramme, dont voici la nouvelle version :

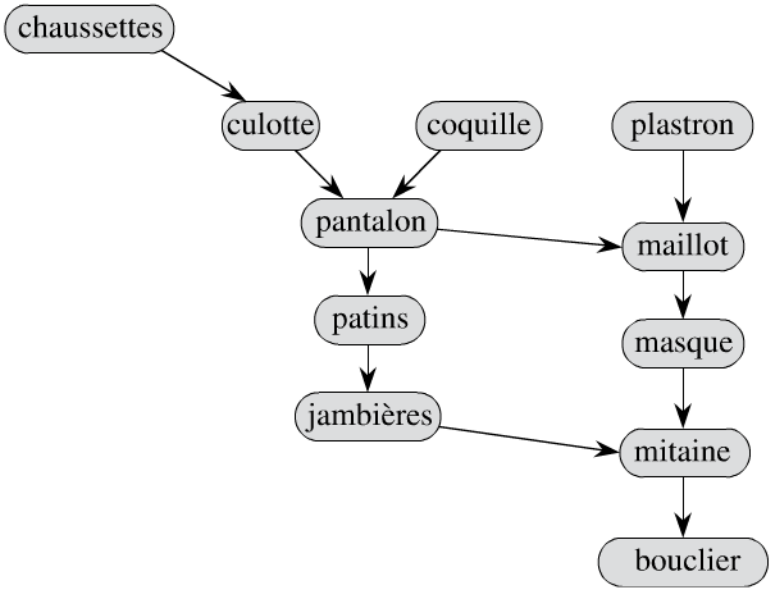



J'ai de nouveau choisi un article sans flèche entrante, cette fois-ci le tee-shirt. Je l'ai ajouté à la fin de l'ordre et retiré du diagramme, arrivant ainsi à la version suivante :

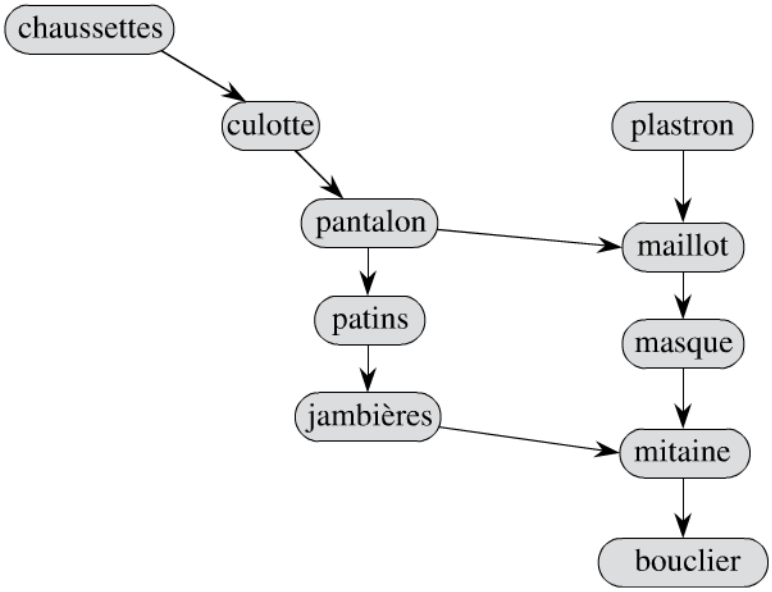



Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

J'ai renouvelé l'opération pour l'article suivant, la gaine, ce qui a conduit au diagramme suivant :




Puis est venu le tour de la coquille :




J'ai poursuivi la procédure « choisir un article sans flèche entrante, l'ajouter à la fin de l'ordre et le retirer du diagramme », jusqu'à ce qu'il ne reste plus aucun article. Les trois ordres donnés au tableau de la page 76 sont issus des différents choix de l'article sans flèche entrante que j'ai pu effectuer à partir du diagramme de la page 76.

## Graphes orientés acycliques

Les diagrammes précédents sont des exemples particuliers de **graphes orientés**. Ils sont composés de **sommets**, qui correspondent aux articles

Copyright © 2013 Dunod.

de l'équipement du gardien de but, et d'**arcs**, représentés par des flèches. Chaque arc est un couple ordonné de la forme $(u,v)$, où $u$ et $v$ sont des sommets. Par exemple, dans le graphe orienté de la page 76, l'arc le plus à gauche est (chaussettes, culotte). Lorsqu'un graphe orienté comprend un arc $(u,v)$, nous disons que $v$ est **adjacent** à $u$ et que $(u,v)$ **sort** de $u$ et **entre** dans $v$. Le sommet libellé culotte est adjacent au sommet libellé chaussettes et l'arc (chaussettes, culotte) sort du sommet chaussettes et entre dans le sommet culotte.

Les graphes orientés que nous avons vus ont une autre propriété : il est impossible d'aller d'un sommet à lui-même en suivant une suite d'un ou plusieurs arcs. Dans ce cas, nous disons qu'il s'agit d'un **graphe orienté acyclique** (DAG, *Directed Acyclic Graph*). Il est acyclique car il n'existe aucun chemin qui forme un circuit (cycle) entre un sommet et lui-même (nous proposerons une définition plus formelle d'un circuit ultérieurement dans ce chapitre).

Les DAG sont parfaitement adaptés à la modélisation des dépendances où une tâche doit être réalisée avant une autre. Ils sont ainsi employés dans la planification des projets, comme la construction d'une maison : les murs doivent être élevés avant de passer au toit. Nous les retrouvons également en cuisine où certaines étapes d'une recette doivent être effectuées dans un ordre précis, tandis que pour d'autres cela n'a pas d'importance (nous verrons un exemple de DAG en cuisine plus loin dans ce chapitre).

## Tri topologique

Si nous devons déterminer un seul ordre linéaire pour enfiler tout l'équipement du gardien de but, nous devons effectuer un « tri topologique ». Plus précisément, le **tri topologique** d'un DAG permet d'obtenir un ordre linéaire tel que si $(u,v)$ est un arc du DAG, alors $u$ apparaît avant $v$ dans l'ordre linéaire. Le tri topologique diffère du tri tel que nous l'avons utilisé aux Chapitres 3 et 4.

L'ordre linéaire obtenu par un tri topologique n'est pas nécessairement unique. Mais vous le saviez déjà, car chacun des trois ordres qui permettent de revêtir l'équipement du gardien de but donnés à la page 76 peut être obtenu par un tri topologique.

Il m'est arrivé d'utiliser le tri topologique dans le cadre d'un travail de programmation. L'objectif était de créer un système de conception assistée par ordinateur, avec une gestion d'une bibliothèque de composants. Les composants pouvaient en contenir d'autres, mais aucune dépendance circulaire n'était autorisée : un composant ne pouvait pas se contenir lui-même. Les composants devaient être enregistrés sur une bande (oui, c'est un travail qui commence à dater) afin que chaque composant précède tout autre composant qui le contenait. Si un composant correspond à un sommet et

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

si un arc $(u, v)$ indique que le composant $v$ contient le composant $u$, alors l'enregistrement des composants pouvait se faire conformément à un ordre linéaire obtenu par tri topologique.

Dans l'ordre linéaire, quel sommet doit venir en premier ? N'importe quel sommet sans arc entrant fera l'affaire. Le nombre d'arcs qui entrent dans un sommet est le **degré entrant**. Nous pouvons donc partir de tout sommet dont le degré entrant est égal à 0. Heureusement, un DAG a nécessairement au moins un sommet de degré entrant égal à 0 et au moins un sommet de **degré sortant** égal à 0 (aucun arc ne sort du sommet) car il existerait sinon un circuit.

Supposons donc que nous choisissions n'importe quel sommet de degré entrant 0, que nous l'appelions sommet $u$ et que nous le placions au début de l'ordre linéaire. Puisque nous avons commencé par le sommet $u$, tous les autres sommets seront placés après $u$ dans l'ordre linéaire. En particulier, tout sommet $v$ adjacent à $u$ doit apparaître quelque part après $u$ dans l'ordre linéaire. Par conséquent, nous pouvons retirer du DAG $u$ et tous les arcs qui en sortent, en sachant que nous avons tenu compte des dépendances définies par cet arc. Lorsque nous retirons un sommet et les arcs qui en sortent, qu'obtenons-nous ? Un autre DAG ! En effet, nous ne pouvons pas créer un circuit en retirant un sommet et des arcs. Nous pouvons ainsi répéter la procédure avec le DAG obtenu : rechercher un sommet de degré entrant 0, le placer après le sommet $u$. Dans l'ordre linéaire, retirer les arcs sortants, etc.

La procédure suivante, qui met en œuvre un tri topologique, se fonde sur cette idée. Cependant, au lieu de supprimer réellement des sommets et des arcs du DAG, nous gérons le degré entrant de chaque sommet, en le décrémentant pour chaque arc retiré conceptuellement. Puisque les indices des tableaux sont des entiers, supposons que nous identifions chaque sommet par un entier unique dans la plage 1 à $n$. Puisque la procédure doit identifier rapidement un sommet de degré entrant 0, elle mémorise le degré entrant de chaque sommet dans un tableau *degré-entrant* indicé par les sommets. Elle gère également une liste *suivants* de tous les sommets ayant un degré entrant égal à 0. Les étapes 1 à 3 initialisent le tableau *degré-entrant*, l'étape 4 initialise *suivants* et l'étape 5 actualise *degré-entrant* et *suivants* lorsque des sommets et des arcs sont conceptuellement supprimés. La procédure peut choisir n'importe quel sommet de la liste *suivants* qu'il faut placer ensuite dans l'ordre linéaire.

*Procédure* TRI-TOPOLOGIQUE($G$)

*Entrée :* $G$, un graphe orienté acyclique dont les sommets sont numérotés de 1 à $n$.

*Sortie* : Un ordre linéaire des sommets dans lequel $u$ apparaît avant $v$ si $(u, v)$ est un arc du graphe.

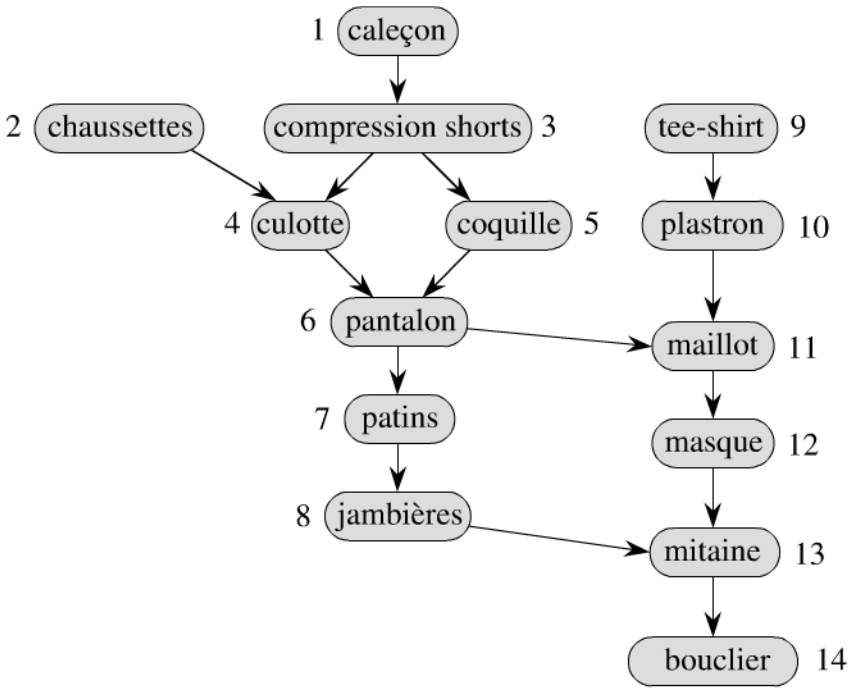
Copyright © 2013 Dunod.

1. Créer le tableau *degré – entrant*[1..*n*] et un ordre linéaire vide de sommets.
2. Fixer toutes les valeurs *degrés-entrant* à 0.
3. Pour chaque sommet *u* :
   A. Pour chaque sommet *v* adjacent à *u* :
      i. Incrémenter *degré – entrant*[*v*].
4. Créer une liste *suivants* de tous les sommets *u* telle que *degré – entrant*[*u*] = 0.
5. Tant que *suivants* n'est pas vide, faire la chose suivante :
   A. Supprimer un sommet de *suivants* et l'appeler sommet *u*.
   B. Ajouter *u* à la fin de l'ordre linéaire.
   C. Pour chaque sommet *v* adjacent à *u* :
      i. Décrémenter *degré – entrant*[*v*].
      ii. Si *degré – entrant*[*v*] = 0, alors insérer *v* dans la liste *suivants*.
6. Retourner l'ordre linéaire.

Examinons les quelques premières itérations de l'étape 5 sur le DAG qui permet d'enfiler l'équipement du gardien. Pour exécuter la procédure TRI-TOPOLOGIQUE sur ce DAG, nous numérotons les sommets conformément au diagramme suivant. Puisque seuls les sommets 1, 2 et 9 ont un degré entrant égal à 0, lorsque nous entrons dans la boucle de l'étape 5, la liste *suivants* comprend uniquement ces trois sommets. Pour obtenir l'ordre n° 1 de la page 76, la liste *suivants* doit contenir les sommets 1, 2, 9 dans cet ordre. Ensuite, dans la première itération de la boucle de l'étape 5, nous choisissons le sommet 1 (caleçon) pour servir de sommet *u*, nous le retirons de *suivants*, nous l'ajoutons à la fin de l'ordre linéaire initialement vide et nous décrémentons *degré – entrant*[3] (gaine). Puisque cette opération amène *degré – entrant*[3] à 0, nous ajoutons le sommet 3 à *suivants*. Supposons que l'insertion d'un sommet dans *suivants* place celui-ci en premier sommet de la liste. Une liste dans laquelle l'ajout et la suppression d'éléments se font toujours depuis la même extrémité se nomme une *pile*. Elle fonctionne en effet à la manière d'une pile d'assiettes, dans laquelle une assiette est toujours prise en haut de la pile et une nouvelle est toujours posée au sommet. Ce type de liste est également désigné par **dernier entré**, **premier sorti** ou **LIFO** (*last in, first out*). Dans ces conditions, *suivants* comprend à présent les sommets 3, 2, 9 et, pour l'itération suivante, nous choisissons le sommet 3 en tant que sommet *u*. Nous le supprimons de *suivants*, l'ajoutons à la fin de l'ordre linéaire, qui devient alors « caleçon, gaine », et décrémentons *degré – entrant*[4] (passe de 2 à 1) et *degré – entrant*[5] (passe de 1 à 0). Nous insérons le sommet 5 (coquille) dans *suivants*, qui contient donc 5, 2, 9. Lors de l'itération

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

suivante, nous prenons le sommet 5 en tant que sommet $u$, le supprimons de *suivants*, l'ajoutons à la fin de l'ordre linéaire, qui devient « caleçon, gaine, coquille », et décrémentons *degré – entrant*[6] (passe de 2 à 1). Cette fois-ci, aucun sommet n'est ajouté à *suivants* et, lors de l'itération suivante, nous choisissons le sommet 2 en tant que sommet $u$, et ainsi de suite.




Pour analyser la procédure TRI-TOPOLOGIQUE, nous devons tout d'abord savoir comment représenter un graphe orienté et une liste comme *suivants*. Le graphe représenté n'est pas nécessairement acyclique, car l'absence ou la présence d'un circuit n'a aucun effet sur sa représentation.

Copyright © 2013 Dunod.

## REPRÉSENTER UN GRAPHE ORIENTÉ

Il existe plusieurs manières de représenter un graphe orienté sur un ordinateur. Selon notre convention, un graphe possède $n$ sommets et $m$ arcs. Nous supposons également que chaque sommet possède son propre numéro, de 1 à $n$, afin qu'il puisse servir d'indice dans un tableau, ou même de numéro de ligne ou de colonne dans une matrice.

Pour le moment, nous souhaitons simplement connaître les sommets et les arcs présents (plus tard, nous associerons également une valeur numérique à chaque arc). Nous pouvons utiliser une **matrice d'adjacence** $n \times n$ dans laquelle chaque ligne et chaque colonne correspondent à un sommet. La cellule à l'intersection de la ligne du sommet $u$ et de la colonne du sommet $v$ vaut 1 si l'arc $(u,v)$ est présent, et 0 sinon. Puisqu'une matrice d'adjacence comprend $n^2$ entrées, il faut que $m \leq n^2$. Une autre solution serait de gérer une liste de tous les arcs $m$ du graphe sans ordre particulier. Nous pouvons également adopter une approche hybride, qui combine la

matrice d'adjacence et la liste non ordonnée. Il s'agit d'une représentation par **liste d'adjacence**, avec un tableau de $n$ éléments indicé par les sommets et dont une entrée pour un sommet $u$ est une liste de tous les sommets adjacents à $u$. Les listes comprennent $m$ sommets car il existe un élément de liste pour chacun des $m$ arcs. Voici les représentations par matrice d'adjacence et par liste d'adjacence du graphe orienté de la page 82 :

**Matrice d'adjacence**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Listes d'adjacence**

| | |
|---|---|
| 1 | 3 |
| 2 | 4 |
| 3 | 4, 5 |
| 4 | 6 |
| 5 | 6 |
| 6 | 7, 11 |
| 7 | 8 |
| 8 | 13 |
| 9 | 10 |
| 10 | 11 |
| 11 | 12 |
| 12 | 13 |
| 13 | 14 |
| 14 | (vide) |

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Nous avons donc une liste non ordonnée d'arcs et une liste d'adjacence, mais comment représente-t-on une liste ? La meilleure solution dépend des opérations qui seront réalisées sur la liste. Pour les listes d'arcs non ordonnées et les listes d'adjacence, nous connaissons par avance le nombre d'arcs présents dans chaque liste et le contenu des listes ne changera pas. Nous pouvons donc stocker chaque liste dans un tableau. Un tableau peut également être employé même si le contenu de la liste évolue avec le temps. Le point critique est de connaître le nombre maximal d'éléments que contiendra la liste à tout moment. Si nous n'avons pas besoin d'insérer ou de supprimer un élément au milieu de la liste, la représentation des listes à l'aide de tableaux est une solution aussi efficace que d'autres.

Si des éléments doivent être insérés au milieu de la liste, nous pouvons nous tourner vers une **liste chaînée**, dans laquelle chaque élément mémorise l'emplacement de son successeur. L'insertion d'un nouvel élément après un élément donné est alors possible. Si nous devons également supprimer un élément au milieu de la liste, alors chacun de ses éléments doit mémoriser l'emplacement de son prédécesseur. À partir de maintenant, nous supposerons que nous avons la possibilité d'insérer et de supprimer en un temps constant des éléments d'une liste chaînée. Si la liste chaînée comprend uniquement des liens vers les successeurs, il s'agit d'une **liste simplement chaînée**. Si elle ajoute des liens vers les prédécesseurs, il s'agit d'une **liste doublement chaînée**.

## Temps d'exécution du tri topologique

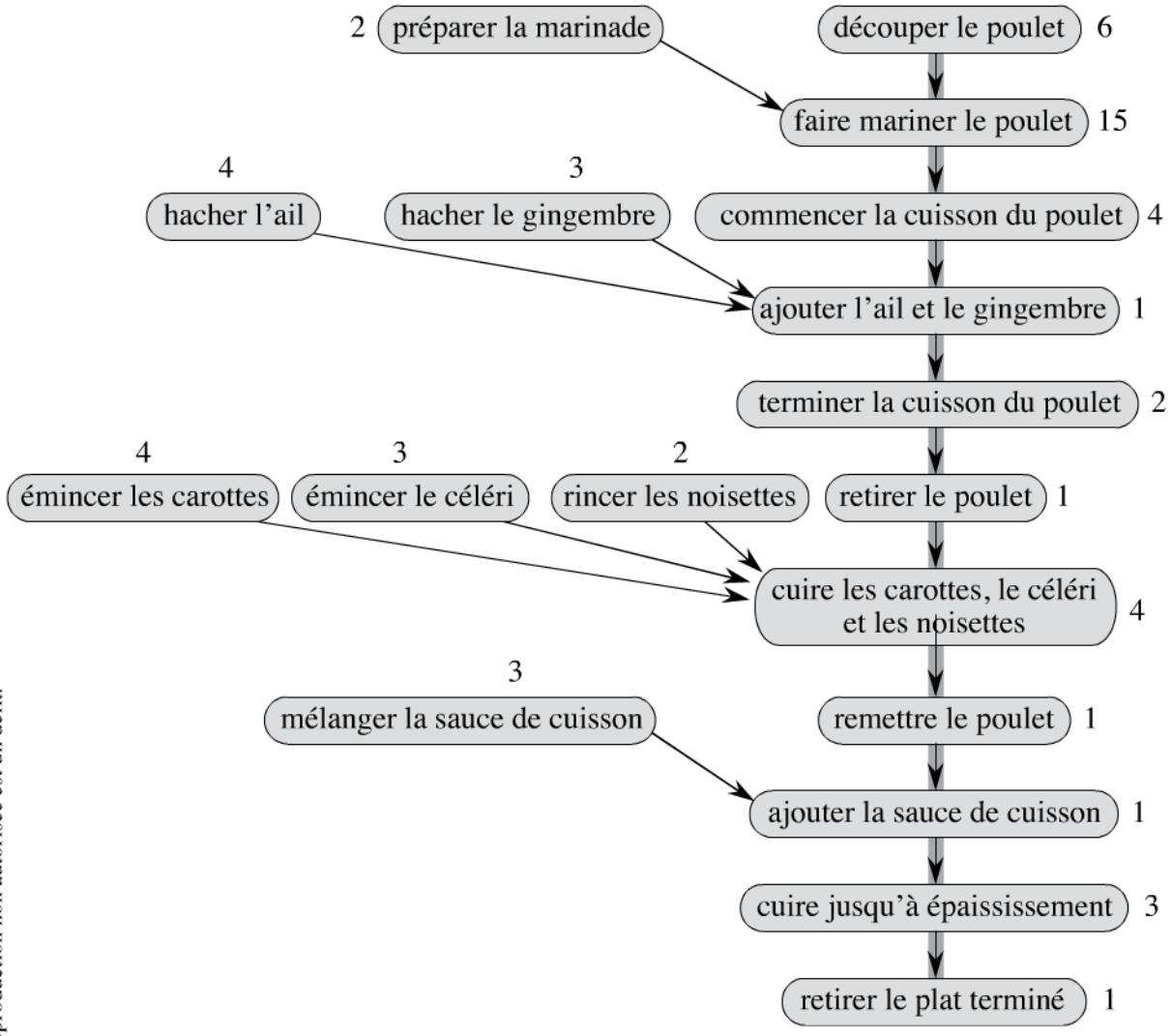
En supposant que le DAG soit représenté sous forme d'une liste d'adjacence et la liste *suivants* sous forme d'une liste chaînée, nous pouvons montrer que la procédure Tri-Topologique prend un temps $\Theta(n + m)$. Puisque *suivants* est une liste chaînée, nous pouvons effectuer des insertions et des suppressions en un temps constant. L'étape 1 se fait en un temps constant et, puisque le tableau *degré-entrant* comprend $n$ éléments, l'étape 2 initialise tous les éléments du tableau à 0 en un temps $\Theta(n)$. L'étape 3 prend un temps $\Theta(n + m)$. Le terme $\Theta(n)$ de l'étape 3 provient de l'analyse des $n$ sommets dans la boucle externe, tandis que le terme $\Theta(m)$ est lié à la visite des $m$ arcs dans la boucle interne de l'étape 3A, une fois pour chaque itération de la boucle externe. Le temps de l'étape 4 est $O(n)$, car la liste *suivants* comprend initialement au plus $n$ sommets. La plus grande partie du travail se fait à l'étape 5. Puisque chaque sommet est inséré dans *suivants* une fois, le nombre d'itérations de la boucle principale est $n$. Les itérations des étapes 5A et 5B prennent un temps constant. À l'image de l'étape 3A, la boucle de l'étape 5C itère à $m$ reprises, une fois par arc. Les étapes 5Ci et 5Cii demandent un temps constant par itération, ce qui conduit à un temps global $\Theta(m)$ pour l'étape 5C. Par conséquent, la boucle

Copyright © 2013 Dunod.

de l'étape 5 se fait en $\Theta(n + m)$. Bien entendu, l'étape 6 a un temps constant et, après que nous avons additionné le temps de toutes les étapes, nous obtenons $\Theta(n + m)$.

## Chemin critique dans un diagramme PERT

Après une journée de travail, j'aime me relaxer en cuisinant, en particulier du poulet kung pao. Je dois découper le poulet, émincer les légumes, préparer une sauce, puis cuire le plat. Comme dans le cas de l'équipement du gardien de but en hockey sur glace, certaines étapes doivent être effectuées avant d'autres. Je peux donc tracer un DAG pour modéliser la préparation de la recette du poulet kung pao ; il est représenté par l'illustration suivante.




Chaque sommet du DAG est accompagné d'un nombre qui précise la durée, en minutes, nécessaire pour réaliser la tâche correspondante. Par exemple, il faut quatre minutes pour hacher l'ail (car j'épluche chaque gousse et j'utilise énormément d'ail). Si vous additionnez les durées de

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

toutes les tâches, comme si elles étaient exécutées en séquence, vous verrez qu'il faut une heure pour préparer un poulet kung pao.

En revanche, avec un commis pour m'aider, deux tâches pourraient être réalisées simultanément. Par exemple, pendant qu'il découperait le poulet, je pourrais préparer la sauce. Avec du personnel supplémentaire, un espace suffisant et des ustensiles en nombre, plusieurs tâches pourraient s'effectuer en même temps. Si nous prenons deux tâches du DAG et déterminons qu'il n'existe aucun chemin qui mène de l'une à l'autre en suivant les flèches, alors chacune d'elles peut être confiée à une personne différente et peut être réalisée en parallèle.

Avec des ressources illimitées (personnel, espace et ustensiles de cuisine) pour effectuer les tâches simultanément, en quelle durée minimale pouvons-nous réaliser un poulet kung pao ? Le DAG est un exemple de **diagramme PERT** (*Program Evaluation and Review Technique*, ou technique d'évaluation et d'examen de programmes). Le temps pour achever le travail, même avec autant de tâches effectuées en parallèle que possible, est donné par le « chemin critique » dans le diagramme PERT. Pour comprendre ce qu'est le chemin critique, nous devons tout d'abord comprendre ce qu'est un chemin.

Dans un graphe, un **chemin** est une suite de sommets et d'arcs qui permettent d'aller d'un sommet à l'autre (ou à lui-même) ; nous disons que le chemin comprend les sommets et les arcs traversés. Par exemple, un des chemins du DAG pour le poulet kung pao est, dans l'ordre, les sommets libellés « hacher l'ail », « ajouter l'ail et le gingembre », « terminer la cuisson du poulet » et « retirer le poulet », sans oublier les arcs qui relient ces sommets. Un chemin qui va d'un sommet à lui-même est un **circuit**, mais, évidemment, un DAG n'en contient pas.

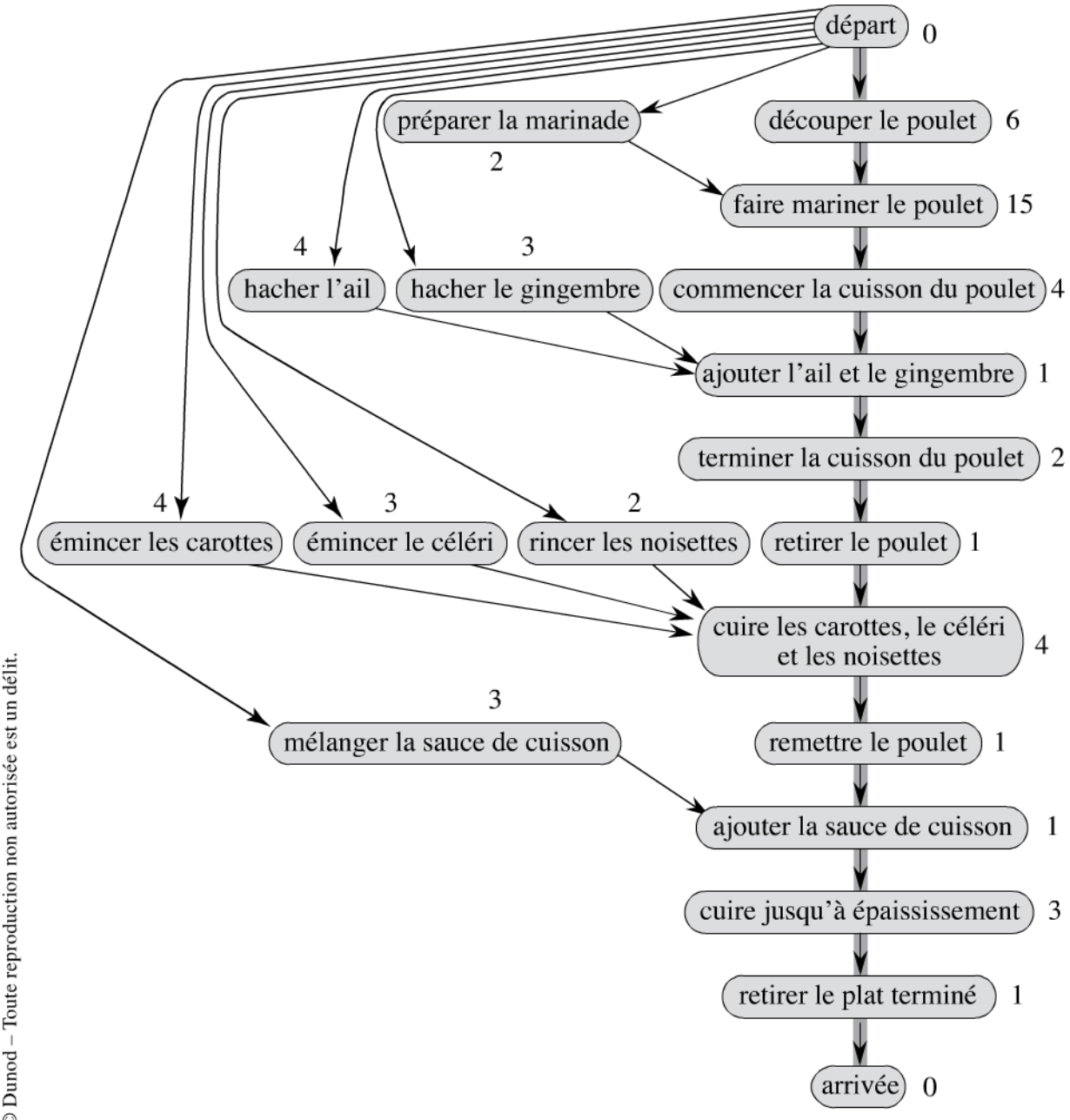
Dans un diagramme PERT, un **chemin critique** correspond à un chemin pour lequel la somme des durées des tâches est maximale sur tous les chemins. La somme des durées le long d'un chemin critique correspond au temps minimal pour l'intégralité de la procédure, quel que soit le nombre de tâches effectuées simultanément. Dans le diagramme PERT de la recette du poulet kung pao, j'ai mis le chemin critique en grisé. Si vous additionnez les durées des tâches rencontrées le long de ce chemin critique, vous constaterez que quel que soit le nombre de personnes pour exécuter la recette, il faudra au moins 39 minutes pour aller au bout[1].

En supposant que les durées de toutes les tâches soient positives, un chemin critique dans un diagramme PERT doit démarrer à un sommet de

1. Si vous vous demandez comment, dans un restaurant chinois, votre assiette de poulet kung pao peut arriver plus rapidement, sachez que la plupart des ingrédients sont préparés à l'avance, sans oublier que leur matériel professionnel est plus performant que le mien.

Copyright © 2013 Dunod.

degré entrant égal à 0 et se terminer à un sommet de degré sortant égal à 0. Au lieu de vérifier les chemins entre tous les couples de sommets dont l'un a un degré entrant égal à 0 et l'autre un degré sortant égal à 0, nous pouvons simplement ajouter des sommets factices, « départ » et « arrivée », comme l'illustre la figure suivante. Puisqu'il s'agit de sommets factices, nous leur affectons des durées égales à 0. Nous ajoutons un arc qui va de départ à chaque sommet de degré entrant 0 dans le diagramme PERT, ainsi qu'un arc qui part de chaque sommet de degré sortant 0 pour atteindre arrivée. À présent, le seul sommet de degré entrant 0 est départ, et le seul sommet de degré sortant 0 est arrivée. Un chemin de départ à arrivée avec la somme maximale des durées des tâches sur ses sommets (en gris) donne un chemin critique dans le diagramme PERT, à l'exception des sommets factices évidemment.




Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

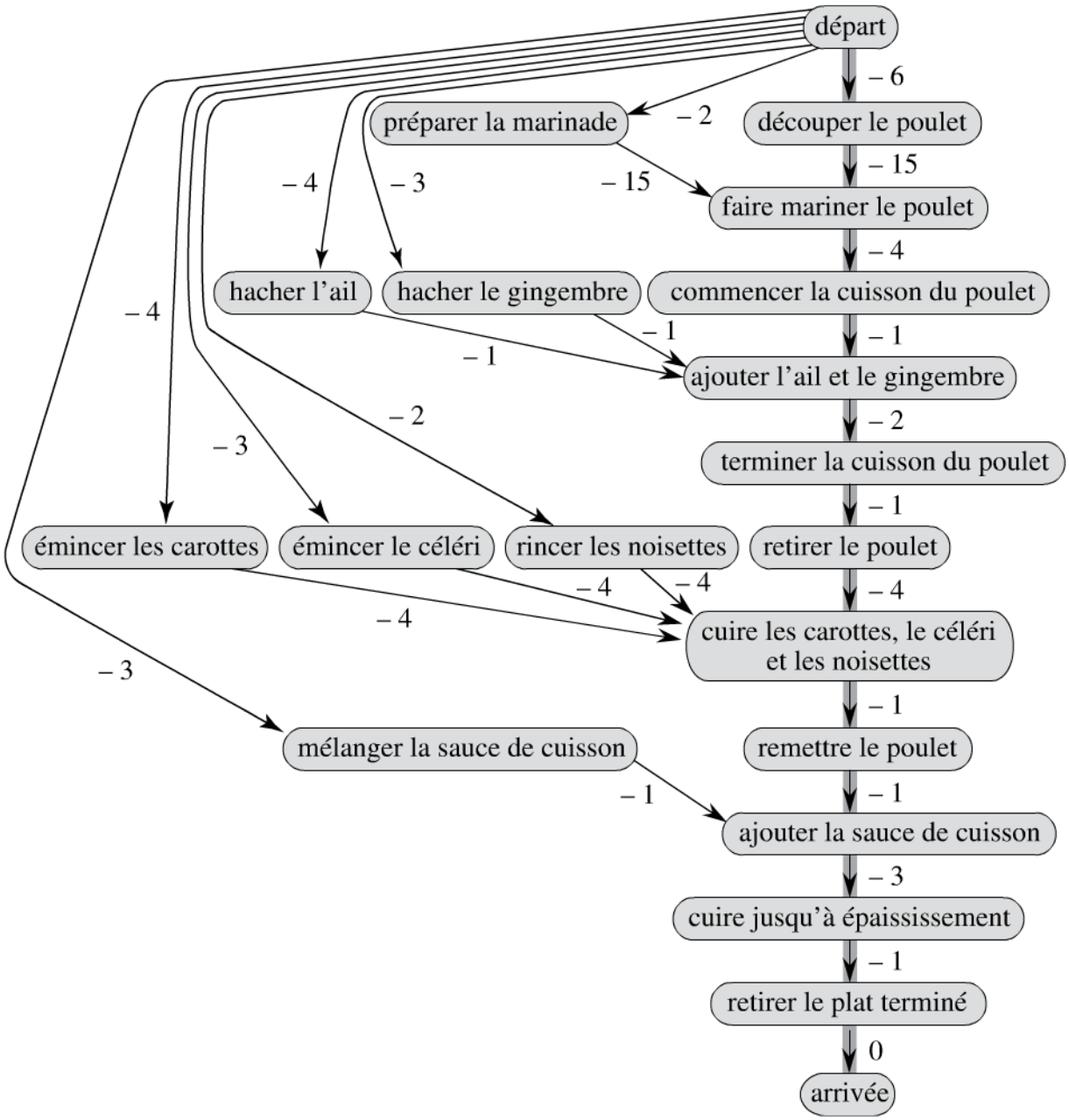
Après que nous avons ajouté les sommets factices, nous trouvons un chemin critique en recherchant le plus court chemin qui va de départ à arrivée, en fonction des durées des tâches. Vous pourriez penser que la phrase précédente est erronée, car un chemin critique doit correspondre au chemin le plus long non au plus court. C'est évidemment le cas, mais puisqu'un diagramme PERT ne contient aucun circuit, nous pouvons modifier les durées des tâches de sorte qu'un plus court chemin donne un chemin critique. Plus précisément, nous inversons chaque durée et recherchons un chemin de départ à arrivée dont la somme des durées est *minimale*.

Pourquoi inverser les durées des tâches et rechercher un chemin avec une somme des durées minimales ? Parce que résoudre ce problème correspond à un cas particulier de la recherche du plus court chemin et que nous disposons d'une multitude d'algorithmes pour cela. Toutefois, dans les problèmes de plus courts chemins, les valeurs qui fixent les longueurs de chemins sont associées aux arcs, non aux sommets. Cette valeur définie sur un arc est appelée son **poids**. Un graphe orienté avec des poids sur les arcs est un **arc orienté pondéré**. « Poids » est un terme générique pour les valeurs associées aux arcs. Si un graphe orienté pondéré représente un réseau routier, chaque arc correspond à une direction de la route entre deux intersections et le poids peut donner la longueur de la route, le temps nécessaire à la parcourir ou le péage à payer pour y circuler. Le **poids d'un chemin** est la somme des poids des arcs qui composent ce chemin. Si les poids des arcs représentent les longueurs des routes, le poids d'un chemin indique la distance totale qui mène d'une extrémité du chemin à l'autre. Un **plus court chemin** depuis le sommet $u$ au sommet $v$ est un chemin dont la somme des poids des arcs est minimale parmi tous les chemins qui vont de $u$ à $v$. Les plus courts chemins ne sont pas nécessairement uniques, car un graphe orienté de $u$ à $v$ peut contenir de nombreux chemins dont la somme des poids est égale au minimum.

Pour transformer un diagramme PERT avec des durées de tâches négatives en un graphe orienté pondéré, nous déplaçons la durée de chaque sommet sur chacun des arcs entrants. Autrement dit, si le sommet $v$ affiche une durée de tâche (non inversée) égale à $t$, nous fixons le poids de chaque arc $(u,v)$ qui entre dans $v$ à $-t$. Le DAG obtenu en appliquant cette conversion est présenté dans la figure suivante (page en vis-à-vis) ; les poids des arcs sont indiqués à côté de chacun d'eux.

À présent, nous devons simplement rechercher un plus court chemin (en gris) qui va de départ à arrivée, en tenant compte des poids des arcs. Dans le diagramme PERT d'origine, un chemin critique correspondra aux sommets du plus court chemin trouvé, hormis les sommets départ et arrivée. Voyons comment trouver un plus court chemin dans un DAG.

Copyright © 2013 Dunod.




# Plus court chemin dans un graphe orienté acyclique

Apprendre à trouver un plus court chemin dans un DAG présente un autre avantage : cela nous permet de poser les fondations de la recherche des plus courts chemins dans des graphes orientés quelconques, qui peuvent inclure des circuits. Nous reviendrons sur ce problème plus général au Chapitre 6. Comme dans le cas du tri topologique d'un DAG, nous supposerons que celui-ci est représenté sous forme d'une liste d'adjacence et qu'avec chaque arc $(u,v)$ est également enregistré son poids donné par $poids(u,v)$.

Dans un DAG obtenu par transformation d'un diagramme PERT, nous voulons trouver un plus court chemin qui va du *sommet source*, appelé

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

« départ », à un **sommet cible** spécifique, appelé « arrivée ». Dans ce cas, nous résolvons le problème plus général de recherche des **plus courts chemins** à origine *unique*, dans lequel nous voulons trouver les plus courts chemins qui partent d'un sommet source et qui vont vers tous les autres sommets. Par convention, le sommet source se nomme $s$, et pour chaque sommet $v$, nous voulons calculer deux choses. Premièrement le poids d'un plus court chemin de $s$ à $v$, que nous notons $pcc(s,v)$. Deuxièmement, le **prédécesseur** de $v$ dans un plus court chemin de $s$ à $v$ : le sommet $u$ tel qu'un plus court chemin de $s$ à $v$ est un chemin de $s$ à $u$ suivi d'un seul arc $(u,v)$. Les $n$ sommets sont numérotés de 1 à $n$ afin que nos algorithmes du plus court chemin de ce chapitre et du Chapitre 6 puissent placer les résultats dans des tableaux $pluscourt[1..n]$ et $pred[1..n]$, respectivement. Pendant le déroulement des algorithmes, les éléments $pluscourt[v]$ et $pred[v]$ n'auront peut-être pas leur valeur finale correcte, mais au terme de l'algorithme, elles le seront.

Nous avons deux cas à traiter. Premièrement, que faire s'il n'existe aucun chemin qui mène de $s$ à $v$ ? Nous définissons $pcc(s,v) = \infty$ de sorte que $pluscourt[v]$ arrive à $\infty$. Puisque $v$ n'aura pas de prédécesseur sur un plus court chemin venant de $s$, nous disons également que $pred[v]$ a la valeur spéciale NULL. Par ailleurs, tous les plus courts chemins venant de $s$ partent de $s$, et $s$ n'a donc pas de prédécesseur ; nous disons que $pred[s]$ doit également être NULL. L'autre cas se rencontre uniquement dans les graphes qui ont des circuits et des arcs avec des poids négatifs : que faire si le poids d'un circuit est négatif ? Nous risquons de parcourir le circuit indéfiniment, en diminuant le poids du chemin à chaque tour. Si nous pouvons aller de $s$ à un circuit négatif et ensuite à $v$, alors $pcc(s,v)$ est indéfini. Toutefois, pour le moment, nous avons uniquement des graphes acycliques, donc sans circuit et encore moins des circuits avec des poids négatifs.

Pour calculer les plus courts chemins à partir d'un sommet $s$ source, nous démarrons avec $pluscourt[s] = 0$ (car nous n'avons nulle part où aller pour passer d'un sommet à lui-même), $pluscourt[v] = \infty$ pour tous les autres sommets $v$ (car nous ne savons pas à l'avance quels sommets sont atteignables à partir de $s$) et $pred[v] = NULL$ pour tous les sommets $v$. Nous appliquons ensuite une suite d'étapes de relâchement aux arcs du graphe :

*Procédure* RELÂCHEMENT$(u,v)$

*Entrées :* $u$ et $v$ : des sommets tels qu'il existe un arc $(u,v)$.

*Résultat :* la valeur de $pluscourt[v]$ peut avoir diminué et, si c'est le cas, alors $pred[v]$ devient $u$.

1. Si $pluscourt[u] + poids(u,v) < pluscourt[v]$, alors fixer $pluscourt[v]$ à $pluscourt[u] + poids(u,v)$ et fixer $pred[v]$ à $u$.

Copyright © 2013 Dunod.

Lorsque nous appelons *Relâchement*$(u,v)$, nous déterminons si nous pouvons améliorer le plus court chemin courant de $s$ à $v$ en prenant $(u,v)$ comme dernier arc. Nous comparons le poids du plus court chemin courant à $u$ plus le poids de l'arc $(u,v)$ au poids du plus court chemin à $v$. S'il est préférable de prendre l'arc $(u,v)$, alors nous actualisons *pluscourt*$[v]$ avec ce nouveau poids et nous fixons le prédécesseur de $v$ sur un plus court chemin à $u$.

Si le relâchement des arcs se fait dans l'ordre le long d'un plus court chemin, nous obtenons des résultats appropriés. Vous vous demandez peut-être comment nous pouvons être assurés que le relâchement des arcs se fera dans l'ordre le long d'un plus court chemin alors que nous ne connaissons pas ce chemin (c'est ce que nous essayons de trouver), mais cela est facile dans un DAG. Nous allons relâcher tous les arcs du DAG et ceux de chaque plus court chemin seront inclus, dans l'ordre, au fur et à mesure que nous passerons par les arcs et que nous relâcherons chacun d'eux.

Voici de façon plus précise comment fonctionne le relâchement des arcs le long d'un plus court chemin. Cela s'applique à n'importe quel graphe orienté, avec ou sans circuits :

> Nous commençons avec *pluscourt*$[u] = \infty$ et *pred*$[u] = NULL$ pour tous les sommets, à l'exception du sommet source $s$ pour lequel *pluscourt*$[s] = 0$.
>
> Ensuite, nous relâchons les arcs le long d'un plus court chemin de $s$ à n'importe quel sommet $v$, dans l'ordre, en commençant par l'arc sortant de $s$ et en terminant par l'arc entrant dans $v$. Des relâchements d'autres arcs peuvent intervenir librement entre ceux effectués le long de ce plus court chemin, mais les valeurs de *pluscourt* ou de *pred* ne peuvent être modifiées que par des relâchements.
>
> Après que les axes ont été relâchés, les valeurs de *pluscourt* et de *pred* pour $v$ sont correctes : *pluscourt*$[v] = pcc(s,v)$ et *pred*$[v]$ est le sommet qui précède $v$ dans un plus court chemin partant de $s$.

Il est assez facile de voir pourquoi le relâchement des arcs le long d'un plus court chemin, dans l'ordre, fonctionne. Supposons qu'un plus court chemin de $s$ à $v$ passe par les sommets $s$, $v_1$, $v_2$, $v_3$,..., $v_k$, $v$, dans cet ordre. Après que l'arc $(s,v_1)$ a été relâché, *pluscourt*$[v_1]$ doit avoir le poids du plus court chemin correct pour $v_1$, et *pred*$[v_1]$ doit être $s$. Après que l'arc $(s,v_2)$ a été relâché, *pluscourt*$[v_2]$ et *pred*$[v_2]$ doivent être corrects. Et ainsi de suite, jusqu'au relâchement de $(v_k,v)$, après quoi *pluscourt*$[v]$ et *pred*$[v]$ ont leur valeur correcte.

Voici de bonnes nouvelles. Dans un DAG, il est relativement facile de relâcher chaque arc exactement une fois tout en relâchant les arcs le long de chaque plus court chemin, dans l'ordre. Comment ? Tout d'abord, il suffit d'appliquer un tri topologique au DAG. Ensuite, nous examinons chaque

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

sommet, pris dans l'ordre linéaire produit par le tri topologique, et nous relâchons tous les arcs qui sortent de ce sommet. Puisque, dans l'ordre linéaire, chaque arc doit sortir d'un sommet avant et entrer dans un sommet ensuite, chaque chemin du DAG doit passer par des sommets dans un ordre conforme à l'ordre linéaire.

*Procédure* DAG-PLUS-COURTS-CHEMINS($G$,$s$)

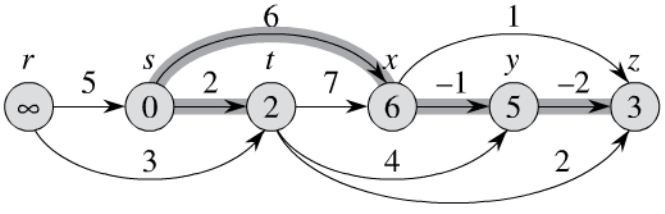
*Entrées* :

- $G$ : un graphe orienté acyclique pondéré qui contient un ensemble $V$ de $n$ sommets et un ensemble $E$ de $m$ arcs.
- $s$ : un sommet source dans $V$.

*Résultat :* Pour chaque sommet $v$ de $V$ autre que le sommet source, *pluscourt*[$v$] est le poids $pcc(s,v)$ d'un plus court chemin de $s$ à $v$ et *pred*[$v$] est le sommet qui précède $v$ sur un plus court chemin. Pour le sommet source $s$, *pluscourt*[$s$] = 0 et *pred*[$s$] = *NULL*. S'il n'existe aucun chemin de $s$ à $v$, alors *pluscourt*[$v$] = $\infty$ et *pred*[$v$] = *NULL*.

1. Appeler TRI-TOPOLOGIQUE($G$) et fixer $l$ à l'ordre linéaire des sommets retourné par l'appel.
2. Fixer *pluscourt*[$v$] à $\infty$ pour chaque sommet $v$, excepté $s$, fixer *pluscourt*[$s$] à 0 et fixer *pred*[$v$] à NULL pour tout sommet $v$.
3. Pour chaque sommet $u$ pris dans l'ordre indiqué par $l$ :
   A. Pour chaque sommet $v$ adjacent à $u$ :
      i. Appeler RELÂCHEMENT($u$,$v$).

L'illustration suivante montre un DAG dans lequel les poids sont placés à côté des arcs. Les valeurs de *pluscourt* obtenues par l'exécution de DAG-PLUS-COURTS-CHEMINS sur le sommet source $s$ sont indiquées à l'intérieur des sommets. Les arcs grisés donnent les valeurs de *pred*. Les sommets sont disposés de gauche à droite selon l'ordre linéaire retourné par le tri topologique. Tous les arcs vont donc de gauche à droite. Si un arc $(u,v)$ est grisé, alors *pred*[$v$] vaut $u$ et *pluscourt*[$v$] = *pluscourt*[$u$] + *poids*($u$,$v$). Par exemple, puisque $(x,y)$ est grisé, *pred*[$y$] = $x$ et *pluscourt*[$y$] (qui vaut 5) est égal à *pluscourt*[$x$] (qui vaut 6) + *poids*($x$,$y$) (qui vaut −1). Puisqu'il n'existe aucun chemin de $s$ à $r$, *pluscourt*[$r$] = $\infty$ et *pred*[$r$] = *NULL* (aucun arc grisé n'entre dans $r$).




Copyright © 2013 Dunod.

La première itération de la boucle de l'étape 3 relâche les arcs $(r,s)$ et $(r,t)$ qui sortent de $r$, mais, puisque $pluscourt[r] = \infty$, ces relâchements n'apportent aucun changement. L'itération suivante relâche les arcs $(s,t)$ et $(s,x)$ qui sortent de $s$, fixant $pluscourt[t]$ à 2, $pluscourt[x]$ à 6 et $pred[t]$ et $pred[x]$ à $s$. L'itération suivante relâche les arcs $(t,x)$, $(t,y)$ et $(t,z)$ qui sortent de $t$. La valeur de $pluscourt[x]$ ne change pas, car $pluscourt[t] + poids(t,x)$, qui vaut $2 + 7 = 9$, est supérieur à $pluscourt[x]$, qui vaut 6. En revanche, $pluscourt[y]$ devient 6, $pluscourt[z]$ devient 4 et $pred[y]$ et $pred[z]$ sont fixés à $t$. L'itération suivante relâche les arcs $(x,y)$ et $(x,z)$ qui sortent de $x$. $pluscourt[y]$ devient alors 5 et $pred[y]$ est fixé à $x$. $pluscourt[z]$ et $pred[z]$ restent inchangés. La dernière itération relâche l'arc $(y,z)$ qui sort de $y$. $pluscourt[z]$ devient 3 et $pred[z]$ est fixé à $y$.

Il est facile de déterminer que DAG-Plus-Courts-Chemins s'exécute en un temps $\Theta(n + m)$. Nous l'avons vu, l'étape 1 prend un temps $\Theta(n + m)$ et, bien entendu, l'étape 2, qui initialise deux valeurs pour chaque sommet, prend un temps $\Theta(n)$. La boucle externe de l'étape 3 examine chaque sommet exactement une fois et la boucle interne de l'étape 3A examine chaque arc exactement une fois pour toutes les itérations. Puisque chaque appel à Relâchement dans l'étape 3Ai se fait en un temps constant, l'étape 3 a un temps $\Theta(n + m)$. Si nous additionnons tous les temps d'exécution des étapes, nous obtenons la durée $\Theta(n + m)$ pour l'ensemble de la procédure.

Revenons aux diagrammes PERT. Nous voyons à présent sans difficulté que rechercher un chemin critique prend un temps $\Theta(n + m)$, avec $n$ sommets et $m$ arcs. Nous ajoutons deux sommets, départ et arrivée, et nous ajoutons au plus $m$ arcs qui sortent de départ et au plus $m$ arcs qui entrent dans arrivée, pour un total maximal de $3m$ arcs dans le DAG. L'inversion des poids et leur déplacement des sommets vers les arcs se font en un temps $\Theta(m)$. Ensuite, rechercher un plus court chemin dans le DAG résultant demande un temps $\Theta(n + m)$.

## Lectures complémentaires

Le Chapitre 22 du CLRS [CLRS09] propose un algorithme de tri topologique d'un DAG différent de celui présenté dans ce chapitre. Vous le retrouverez également dans le volume 1 de *The Art of Computer Programming* par Knuth [Knu97]. La méthode décrite dans le CLRS est au premier abord un tantinet plus simple, mais elle est moins intuitive que celle de ce chapitre et elle se fonde sur une technique de visite des sommets d'un graphe appelée « recherche en profondeur ». L'algorithme de recherche des plus courts chemins à partir d'une origine unique dans un DAG est donné au Chapitre 24 du CLRS.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Vous trouverez de plus amples informations sur les diagrammes PERT, employés depuis les années 1950, dans n'importe quel ouvrage qui traite de la gestion de projet.

Copyright © 2013 Dunod.

# 6 PLUS COURTS CHEMINS

Au Chapitre 5, nous avons vu comment rechercher les plus courts chemins à partir d'une source unique dans un graphe orienté acyclique. Cet algorithme supposait que le graphe était acyclique, c'est-à-dire sans circuit, afin que ses sommets puissent tout d'abord être triés de façon topologique.

Cependant, la plupart des graphes qui modélisent des situations réelles présentent des circuits. Par exemple, dans le graphe d'un réseau routier, chaque sommet représente une intersection et chaque arc représente une route entre deux intersections que nous pouvons emprunter dans une direction. (Les routes à deux voies seraient représentées par deux arcs distincts, allant dans le sens opposé.) Il faut que de tels graphes comprennent des circuits, sinon après avoir quitté une intersection, nous ne pourrions plus y retourner. Lorsque notre GPS calcule l'itinéraire le plus court ou le plus rapide vers une destination, il manipule donc un graphe qui comprend des circuits, en grande quantité.

Lorsque notre GPS recherche l'itinéraire le plus rapide entre notre lieu actuel et une destination précise, il résout le problème de **plus chemin court à couple unique**. Pour cela, il emploie probablement un algorithme qui recherche tous les plus courts chemins à origine unique, mais il retient uniquement celui qui mène à la destination indiquée.

Le GPS travaille avec un graphe orienté pondéré dans lequel les poids des arcs représentent une distance ou une durée. Puisqu'il est impossible de conduire sur une distance négative ou d'arriver avant d'être parti, tous les poids sont positifs. Puisque nous pouvons supposer que, pour une raison quelconque, les poids de certains arcs soient égaux à 0, nous disons que les poids sont positifs. Dans ce cas, nous n'avons pas à nous préoccuper des circuits de poids négatifs et tous les plus courts chemins sont parfaitement définis.

Prenons pour autre exemple de plus court chemin à origine unique le jeu intitulé Les Six degrés de Kevin Bacon : le joueur doit relier des acteurs de cinéma à Kevin Bacon. Dans le graphe, chaque sommet représente un

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

acteur et des arcs $(u,v)$ et $(v,u)$ existent si les acteurs représentés par les sommets $u$ et $v$ sont apparus dans le même film. Étant donné un acteur, le joueur tente de trouver le plus court chemin qui part du sommet de cet acteur et qui arrive au sommet de Kevin Bacon. Le nombre d'arcs dans le plus court chemin (autrement dit le poids du plus court chemin lorsque le poids de chaque arc vaut 1) donne le « nombre de Bacon » de l'acteur. Par exemple, Renée Adorée a joué dans un film avec Bessie Love, qui a joué dans un film avec Eli Wallach, qui a fait un film avec Kevin Bacon. Par conséquent, le nombre de Bacon de Renée Adorée est 3. Les mathématiciens ont un concept similaire nommé nombre d'Erdõs, qui correspond au plus court chemin entre l'éminent Paul Erdõs et n'importe quel autre mathématicien en suivant une chaîne de relations de type coauteur[1].

Quid des graphes avec des arcs dont les poids sont négatifs ? Quel est leur lien avec le monde réel ? Nous verrons que nous pouvons reformuler la question de déterminer si une opportunité d'arbitrage existe dans des échanges monétaires en déterminer si un graphe qui peut avoir des arcs de poids négatifs comprend un circuit de poids négatif.

Du côté des algorithmes, nous commencerons par étudier celui de Dijkstra, qui permet de trouver le plus court chemin entre un sommet source unique et tous les autres sommets. Cet algorithme opère sur des graphes qui présentent deux différences importantes par rapport à ceux du Chapitre 5 : tous les arcs doivent avoir des poids positifs et le graphe peut comprendre des circuits. Vous le trouverez au cœur de la solution mise en œuvre par votre GPS pour déterminer les itinéraires. Nous verrons également les différents choix possibles dans son implémentation. Ensuite, nous aborderons l'algorithme de Bellman-Ford, qui constitue une méthode remarquablement simple pour trouver les plus courts chemins à origine unique même lorsque le graphe comprend des arcs aux poids négatifs. Nous pouvons utiliser les résultats de cet algorithme pour déterminer si le graphe contient un circuit de poids négatif et, dans l'affirmative, pour identifier ses sommets et ses arcs. Puisque les algorithmes de Dijkstra et de Bellman-Ford datent de la fin des années 1950, ils ont passé avec succès l'épreuve du temps. Nous terminerons par l'algorithme de Floyd-Warshall, qui résout le problème des plus courts chemins pour tout couple de sommets.

Comme nous l'avons fait au Chapitre 5 pour la recherche des plus courts chemins dans un DAG, nous supposons que nous avons un sommet source $s$ et le poids $poids(u,v)$ de chaque arc $(u,v)$, et que nous souhaitons calculer, pour chaque sommet $v$, le poids du plus court chemin $pcc(s,v)$ de $s$ à $v$, ainsi que le sommet qui précède $v$ sur un certain plus court chemin partant

1. Croyez-le ou non, il existe même un nombre d'Erdõs-Bacon, qui correspond à la somme des nombres d'Erdõs et de Bacon. De nombreuses personnes ont un nombre d'Erdõs-Bacon fini, notamment Paul Erdõs lui-même !

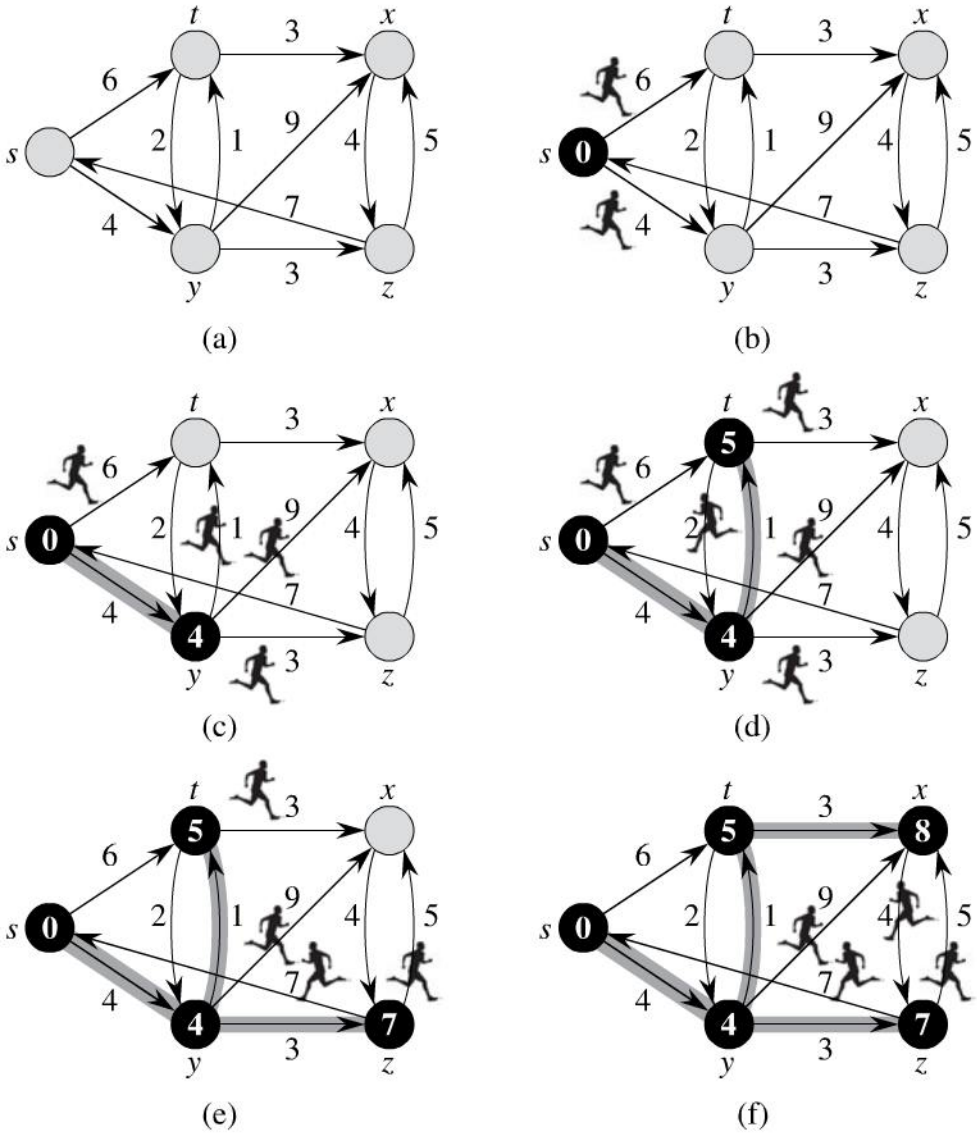
Copyright © 2013 Dunod.

de $s$. Les résultats seront mémorisés dans *pluscourt*[$v$] et dans *pred*[$v$], respectivement.

## Algorithme de Dijkstra

J'aime aborder l'algorithme de Dijkstra[1] comme une simulation de l'envoi de coursiers sur le graphe.

Idéalement, la simulation fonctionne de la manière suivante, bien que j'aie déjà vu l'algorithme de Dijkstra opérer de façon légèrement différente. Il commence par envoyer des coursiers à partir du sommet source vers tous les sommets adjacents. La première fois qu'un coursier arrive sur un sommet, des coursiers partent immédiatement de ce sommet en direction des sommets adjacents.




1. Ce nom vient d'Edsger Dijkstra, qui a proposé l'algorithme en 1959.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Examinons la partie (a) de la figure précédente. Elle montre un graphe orienté avec un sommet source $s$ et des poids attribués aux arcs. Chaque poids représente le nombre de minutes qu'il faut pour que le coursier parcoure l'arc associé.

La partie (b) illustre le début de la simulation, à l'instant 0. À ce moment-là, indiqué à l'intérieur du sommet $s$, des coursiers quittent $s$ en direction de ses deux sommets adjacents, $t$ et $y$. Le sommet $s$ noirci indique que nous savons que $pluscourt[s] = 0$.

Quatre minutes plus tard, à l'instant 4, le coursier correspondant arrive au sommet $y$, comme l'illustre la partie (c). Puisque ce coursier est le premier à atteindre $y$, nous savons que $pluscourt[y] = 4$ et $y$ est donc noirci. L'arc grisé $(s, y)$ indique que le premier coursier à arriver au sommet $y$ vient du sommet $s$ et donc que $pred[y] = s$. À l'instant 4, le coursier qui va du sommet $s$ au sommet $t$ n'est pas encore arrivé. D'autres coursiers quittent le sommet $y$ à l'instant 4, en direction des sommets $t$, $x$ et $z$.

L'événement suivant, représenté en partie (d), se produit une minute plus tard, à l'instant 5, lorsque le coursier parti de $y$ arrive au sommet $t$. Le coursier parti de $s$ vers $t$ n'est pas encore arrivé. Puisque le premier coursier à arriver au sommet $t$ est parti du sommet $y$ à l'instant 5, nous fixons $pluscourt[t]$ à 5 et $pred[t]$ à $y$ (indiqué par l'arc grisé $(y,t)$). Des coursiers quittent le sommet $t$, en direction des sommets $x$ et $y$.

Le coursier parti du sommet $s$ arrive finalement au sommet $t$ à l'instant 6, mais le coursier du sommet $y$ est déjà passé une minute auparavant et les efforts du coursier de $s$ vers $t$ ont donc été vains.

À l'instant 7, illustré en partie (e), deux coursiers arrivent à leur destination. Celui qui vient du sommet $t$ arrive au sommet $y$, mais le coursier parti de $s$ est déjà arrivé à l'instant 4 et la simulation oublie donc le coursier envoyé de $t$ à $y$. Au même moment, le coursier parti de $y$ arrive au sommet $z$. Nous fixons $pluscourt[z]$ à 7 et $pred[z]$ à $y$. Des coursiers partent du sommet z, en direction des sommets $s$ et $x$.

L'événement suivant se produit à l'instant 8, comme illustré en partie (f). Le coursier parti du sommet $t$ arrive au sommet $x$. Nous fixons $pluscourt[x]$ à 8 et $pred[x]$ à $t$, et un coursier quitte le sommet $x$ pour se rendre en $z$.

À ce stade, un coursier est arrivé à chaque sommet et la simulation peut s'arrêter. Tous les coursiers encore en déplacement arriveront à leur sommet de destination après qu'un autre coursier y soit déjà passé. Dès que chaque sommet a vu arriver un coursier, la valeur de *pluscourt* pour chaque sommet est égale au poids du plus court chemin à partir du sommet $s$, et celle de *pred* pour chaque sommet donne le prédécesseur sur un plus court chemin issu de $s$.

Voilà comment fonctionne idéalement la simulation. Le temps mis par un coursier pour parcourir un arc est donné par le poids associé à cet arc.

Copyright © 2013 Dunod.

L'algorithme de Dijkstra opère de façon légèrement différente. Il considère que tous les arcs sont égaux et, lorsqu'il examine ceux qui partent d'un sommet, il traite donc les sommets adjacents ensemble, sans ordre précis. Par exemple, lorsque l'algorithme de Dijkstra traite les arcs qui partent du sommet $s$ de la figure de la page 97, il déclare que $pluscourt[y] = 4$, $pluscourt[t] = 6$ et $pred[y]$ et $pred[t]$ valent tous deux $s$, jusque-là. Lorsqu'il examine ensuite l'arc $(y,t)$, il diminue le poids du plus court chemin vers le sommet $t$ qu'il a déterminé jusque-là et $pluscourt[t]$ passe donc de 6 à 5 et $pred[t]$ devient $y$.

L'algorithme de Dijkstra appelle la procédure RELÂCHEMENT de la page 90, une fois pour chaque arc. Le relâchement d'un arc $(u,v)$ correspond à l'arrivée d'un coursier depuis le sommet $u$ au sommet $v$. L'algorithme gère un ensemble $Q$ de sommets pour lesquels les valeurs finales de *pluscourt* et de *pred* ne sont pas encore connues ; tous les sommets qui ne sont pas présents dans $Q$ possèdent leurs valeurs finales dans *pluscourt* et *pred*. Après l'initialisation de $pluscourt[s]$ à 0 pour le sommet source $s$, de $pluscourt[v]$ à $\infty$ pour tous les autres sommets et de $pred[v]$ à NULL pour tous les sommets, il recherche de façon répétée dans $Q$ le sommet $u$ dont la valeur de *pluscourt* est la plus faible, retire ce sommet de $Q$ et relâche tous les arcs qui sortent de $u$. Voici la procédure correspondante :

*Procédure* DIJKSTRA$(G,s)$

*Entrées :*

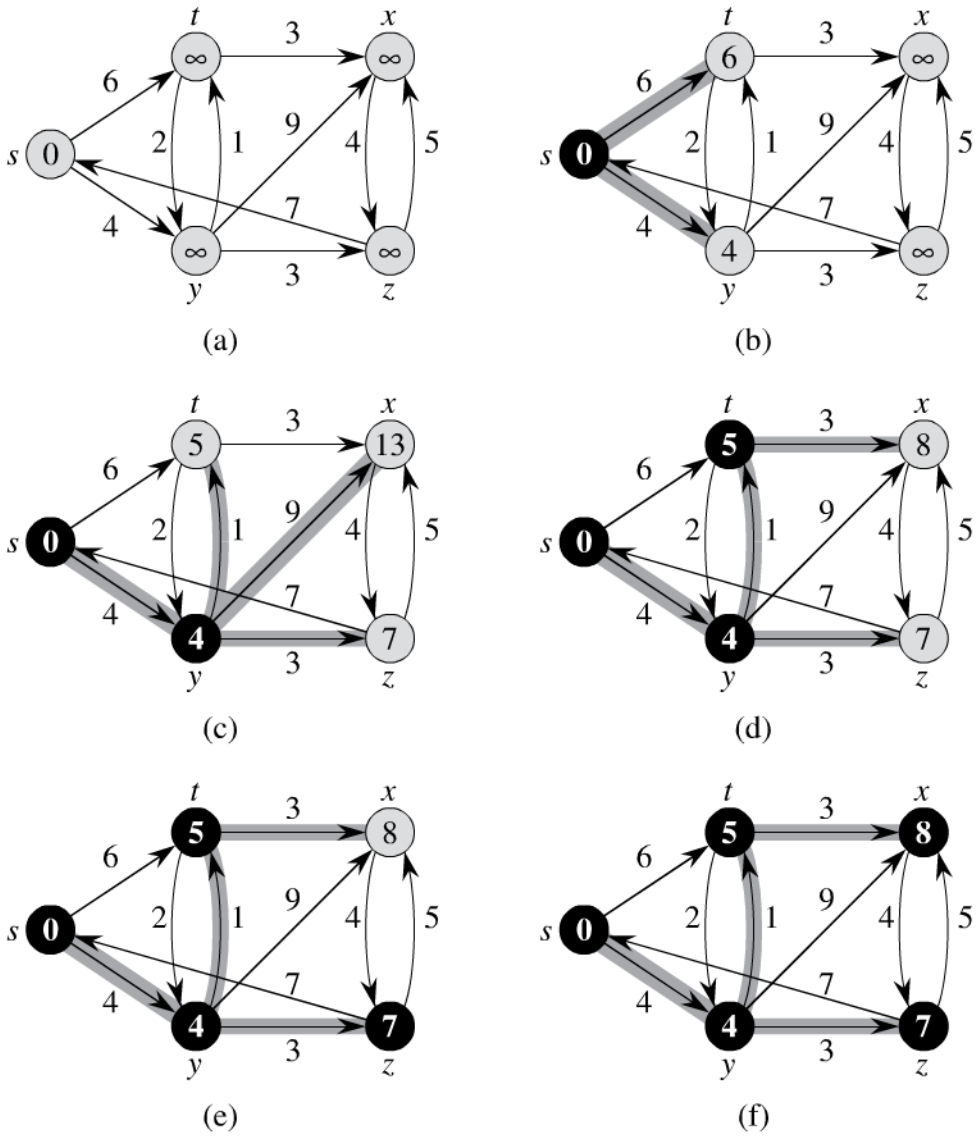
- $G$ : un graphe orienté qui contient un ensemble $V$ de $n$ sommets et un ensemble $E$ de $m$ arcs avec des poids positifs.
- $s$ : un sommet source dans $V$.

*Résultat :* Pour chaque sommet non source $v$ dans $V$, $pluscourt[v]$ est le poids $pcc(s,v)$ d'un plus court chemin de $s$ à $v$ et $pred[v]$ est le sommet qui précède $v$ sur un plus court chemin. Pour le sommet source $s$, $pluscourt[s] = 0$ et $pred[s] = NULL$. S'il n'existe aucun chemin de $s$ à $v$, alors $pluscourt[v] = \infty$ et $pred[v] = NULL$. (Résultat identique à DAG-PLUS-COURTS-CHEMINS à la page 92.)

1. Fixer $pluscourt[v]$ à $\infty$ pour chaque sommet $v$, excepté $s$, fixer $pluscourt[s]$ à 0 et fixer $pred[v]$ à NULL pour tout sommet $v$.
2. Créer $Q$ avec tous les sommets.
3. Tant que $Q$ n'est pas vide, faire la chose suivante :
   A. Rechercher dans $Q$ le sommet $u$ pour lequel la valeur de *pluscourt* est la plus faible et le retirer de $Q$.
   B. Pour chaque sommet $v$ adjacent à $u$ :
      i. Appeler RELÂCHEMENT$(u,v)$.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Chaque partie de la figure suivante montre la valeur de *pluscourt* (indiquée dans chaque sommet), la valeur de *pred* (indiquée par les arcs grisés) et l'ensemble $Q$ (les sommets qui sont grisés mais non noircis) juste avant chaque itération de la boucle de l'étape 3.




Le nouveau sommet noirci dans chaque partie de la figure est celui qui a été choisi en tant que sommet $u$ à l'étape 3A. Dans la simulation avec les coursiers, dès qu'un sommet reçoit ses valeurs *pluscourt* et *pred*, elles ne peuvent plus changer. En revanche, dans cette version, un sommet peut recevoir des valeurs *pluscourt* et *pred* suite au relâchement d'un arc, mais elles peuvent ensuite être modifiées par le relâchement ultérieur d'un autre arc. Par exemple, après que l'arc $(y,x)$ a été relâché en partie (c) de la figure, la valeur de $pluscourt[x]$ décroît de $\infty$ à 13 et $pred[x]$ devient $y$. L'itération suivante de la boucle de l'étape 3 (partie (d)) relâche l'arc $(t,x)$ et $pluscourt[x]$ décroît encore, à 8, et $pred[x]$ devient $t$. Lors de l'itération suivante (partie (e)), l'arc $(z,x)$ est relâché, mais cette fois-ci $pluscourt[x]$

Copyright © 2013 Dunod.

ne change pas car sa valeur, 8, est inférieure à $pluscourt[z] + poids(z,x)$, qui vaut 12.

L'algorithme de Dijkstra conserve l'invariant de boucle suivant :

> Au début de chaque itération de la boucle de l'étape 3, $pluscourt[v] = pcc(s,v)$ pour chaque sommet $v$ qui n'appartient pas à l'ensemble $Q$. Autrement dit, pour chaque sommet $v$ absent de $Q$, la valeur de $pluscourt[v]$ est le poids d'un plus court chemin de $s$ à $v$.

Voici une version simplifiée du raisonnement derrière cet invariant de boucle (une preuve formelle serait un peu plus complexe). Initialement, tous les sommets sont dans l'ensemble $Q$ et l'invariant de boucle ne s'applique à aucun sommet jusqu'à la première itération de la boucle à l'étape 3. Supposons qu'à l'entrée dans une itération de cette boucle, tous les sommets qui ne sont pas dans $Q$ aient leur poids de plus court chemin correct dans *pluscourt*. Dans ce cas, chaque arc qui sort de ces sommets a été relâché au cours d'une exécution de l'étape 3Bi. Prenons le sommet $u$ de $Q$ dont la valeur *pluscourt* est la plus faible. Celle-ci ne pourra jamais être inférieure. En effet, les seuls arcs qui doivent encore être relâchés sont ceux qui sortent des sommets présents dans $Q$ et chacun de ces sommets a une valeur *pluscourt* au moins aussi élevée que $pluscourt[u]$. Puisque les poids de tous les arcs sont positifs, nous devons avoir $pluscourt[u] \leq pluscourt[v] + poids(v,u)$ pour chaque sommet $v$ dans $Q$, et aucune étape de relâchement ultérieure ne diminuera $pluscourt[u]$. Par conséquent, $pluscourt[u]$ a sa valeur la plus faible possible et nous pouvons retirer le sommet $u$ de l'ensemble $Q$ et relâcher tous les arcs qui sortent de $u$. Lorsque la boucle de l'étape 3 se termine, l'ensemble $Q$ est vide et tous les sommets possèdent donc leur poids de plus court chemin correct dans *pluscourt*.

Nous pouvons commencer l'analyse du temps d'exécution de la procédure DIJKSTRA, mais, pour la mener à bien, nous devons tout d'abord faire des choix d'implémentation. Au Chapitre 5, nous avons appelé $n$ le nombre de sommets et $m$ le nombre d'arcs, avec $m \leq n^2$. Nous savons que l'étape 1 nécessite un temps $\Theta(n)$. Nous savons également que la boucle de l'étape 3 effectue $n$ itérations, car l'ensemble $Q$ contient initialement les $n$ sommets, chaque itération de la boucle retire un sommet de $Q$ et les sommets ne sont jamais replacés dans $Q$. La boucle de l'étape 3A traite chaque sommet et chaque arc exactement une fois sur l'ensemble de l'algorithme (nous avons vu le même principe dans les procédures TRI-TOPOLOGIQUE et DAG-PLUS-COURTS-CHEMINS au Chapitre 5).

Que reste-t-il à analyser ? Nous devons déterminer le temps nécessaire à placer les $n$ sommets dans l'ensemble $Q$ (étape 2), celui nécessaire à déterminer le sommet de $Q$ qui possède la valeur *pluscourt* la plus faible et pour le retirer de $Q$ (étape 3A), et celui nécessaire aux quelques ajustements à

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

réaliser, s'ils existent, lorsque les valeurs *pluscourt* et *pred* d'un sommet changent suite à l'appel de RELÂCHEMENT. Nommons ces opérations :

- INSÉRER($Q$,$v$) insère le sommet $v$ dans l'ensemble $Q$. L'algorithme de Dijkstra appelle INSÉRER à $n$ reprises.
- EXTRAIRE-MIN($Q$) retire de $Q$ le sommet qui a la valeur *pluscourt* minimale et retourne ce sommet à l'appelant. L'algorithme de Dijkstra appelle EXTRAIRE-MIN à $n$ reprises.
- DIMINUER-CLÉ($Q$,$v$) réalise toutes les opérations nécessaires sur $Q$ afin de mémoriser que *pluscourt*[$v$] a été diminué par un appel à RELÂCHEMENT. L'algorithme de Dijkstra appelle DIMINUER-CLÉ jusqu'à $m$ fois.

Réunies, ces trois opérations définissent une **file de priorité**.

Les descriptions des opérations de la file de priorité précisent uniquement leur rôle, non leur mise en œuvre. En conception logicielle, la séparation du rôle des opérations et de leur mise en œuvre se nomme **abstraction**. Nous appelons l'ensemble des opérations, précisées par leur rôle non par leur implémentation, un **type de données abstrait** (ADT, *Abstract Data Type*). Une file de priorité est donc un **ADT**.

Nous pouvons implémenter les opérations de la file de priorité en utilisant diverses structures de données. Une **structure de données** correspond à une façon d'enregistrer et d'accéder à des données sur un ordinateur. Par exemple, un tableau est une structure de données. Dans le cas des files de priorité, nous verrons trois structures de données différentes qui permettent de mettre en œuvre leurs opérations. Les concepteurs de logiciels doivent être capables d'employer n'importe quelle structure de données qui implémentent les opérations d'un ADT. Toutefois, du point de vue des algorithmes, cela n'est pas aussi simple. En effet, selon la structure de données, la manière d'implémenter les opérations peut conduire à des temps d'exécution différents. Bien entendu, les trois structures de données que nous utiliserons pour l'implémentation de la file de priorité conduisent à des temps d'exécution de l'algorithme de Dijkstra différents.

Une version revue de la procédure DIJKSTRA, qui appelle explicitement les opérations de la file de priorité, est donnée ci-après. Examinons les trois structures de données qui permettent d'implémenter les opérations de la file de priorité et voyons comment elles affectent le temps d'exécution de cet algorithme.

*Procédure* DIJKSTRA($G$,$s$)

*Entrées et résultat* : Identiques à la version précédente.

1. Fixer *pluscourt*[$v$] à $\infty$ pour chaque sommet $v$, excepté $s$, fixer *pluscourt*[$s$] à 0 et fixer *pred*[$v$] à NULL pour tout sommet $v$.
2. Créer $Q$ sous forme de file de priorité vide.

Copyright © 2013 Dunod.

3. Pour chaque sommet $v$ :
   A. Appeler INSÉRER$(Q, v)$.
4. Tant que $Q$ n'est pas vide, faire la chose suivante :
   A. Appeler EXTRAIRE-MIN$(Q)$ et affecter à $u$ le sommet retourné.
   B. Pour chaque sommet $v$ adjacent à $u$ :
      i. Appeler RELÂCHEMENT$(u, v)$.
      ii. Si l'appel à RELÂCHEMENT$(u, v)$ a diminué la valeur de *pluscourt*$[v]$, appeler DIMINUER-CLÉ$(Q, v)$.

## Implémentation avec un tableau simple

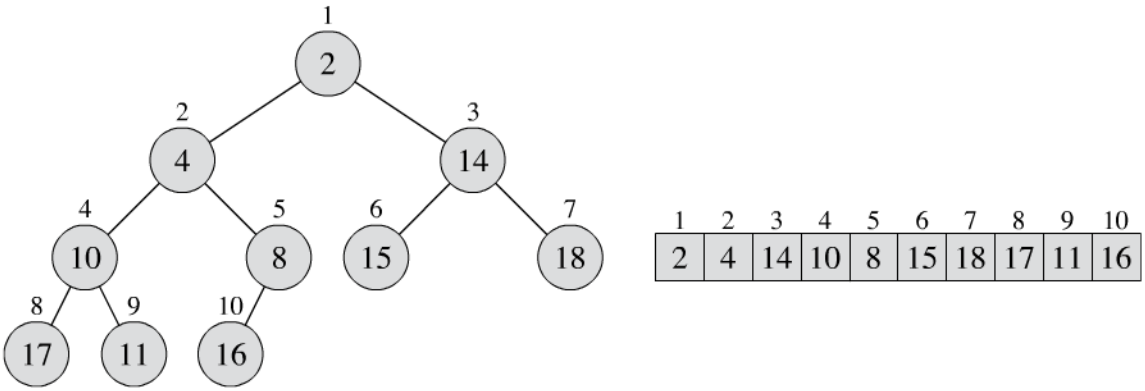
La façon la plus simple d'implémenter les opérations de la file de priorité consiste à stocker les sommets dans un tableau à $n$ emplacements. Si la file de priorité contient actuellement $k$ sommets, ils se trouvent dans les $k$ premiers emplacements du tableau, sans ordre précis. En parallèle du tableau, nous gérons un compteur qui indique le nombre de sommets actuellement présents dans le tableau. L'opération INSÉRER est simple : il suffit d'ajouter le sommet à l'emplacement suivant inutilisé du tableau et d'incrémenter le compteur. L'opération DIMINUER-CLÉ est encore plus simple : il suffit de ne rien faire ! Ces deux opérations se font en un temps constant. En revanche, le temps d'exécution de l'opération EXTRAIRE-MIN est $O(n)$ car nous devons examiner tous les sommets actuellement présents dans le tableau à la recherche de celui dont la valeur *pluscourt* est la plus faible. Après que nous avons identifié ce sommet, sa suppression est facile : il suffit de déplacer le sommet du dernier emplacement vers l'emplacement du sommet supprimé et de décrémenter le compteur. Les $n$ appels à EXTRAIRE-MIN prennent un temps $O(n^2)$. Les appels à RELÂCHEMENT demandent un temps $O(m)$, mais $m \leq n^2$. Dans cette implémentation de la file de priorité, l'algorithme de Dijkstra a donc un temps d'exécution $O(n^2)$, principalement dû aux appels à EXTRAIRE-MIN.

## Implémentation avec un tas binaire

Un tas binaire organise ses données sous forme d'un arbre binaire stocké dans un tableau. Un **arbre binaire** est un type de graphe, mais les sommets sont appelés **nœuds**, les arcs ne sont pas orientés et chaque nœud comprend 0,1 ou 2 nœuds sous lui, qui sont ses **enfants**. La partie gauche de la figure suivante illustre un arbre binaire dont les nœuds sont numérotés. Les nœuds qui n'ont pas d'enfant, comme les nœuds 6 à 10, sont des **feuilles**[1].

1. Les informaticiens trouvent qu'il est plus commode de dessiner les arbres en plaçant la racine en haut, avec des branches qui vont vers le bas plutôt que de les dessiner comme des arbres réels, avec la racine en bas et les branches vers le haut.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.




Un **tas binaire** est un arbre binaire qui possède trois propriétés supplémentaires. Premièrement, tous les niveaux de l'arbre sont remplis, à l'exception éventuelle du dernier, qui peut être partiellement rempli en partant de la gauche. Deuxièmement, chaque nœud contient une clé (indiquée à l'intérieur du nœud sur la figure). Troisièmement, les clés respectent la **propriété de tas** : la clé de chaque nœud est inférieure ou égale aux clés de ses enfants. L'arbre binaire de la figure est aussi un tas binaire.

Un tas binaire peut être stocké dans un tableau, comme l'illustre la partie droite de la figure. En raison de la propriété de tas, le nœud qui possède la clé minimale doit toujours se trouver à l'emplacement 1. Les enfants du nœud à l'emplacement $i$ se trouvent aux emplacements $2i$ à $2i + 1$, et son **parent** se trouve à l'emplacement $\lfloor i / 2 \rfloor$. Lorsqu'un tas binaire est stocké dans un tableau, les déplacements vers le haut et le bas sont plus faciles.

Un tas binaire présente également une autre caractéristique importante : s'il comprend $n$ nœuds, alors sa **hauteur**, c'est-à-dire le nombre d'arêtes à partir de la racine jusqu'à la feuille la plus éloignée, est uniquement $\lfloor \lg n \rfloor$. Par conséquent, nous pouvons parcourir un chemin depuis la racine vers une feuille ou depuis une feuille la racine en un temps $O(\lg n)$.

Puisque la hauteur d'un tas binaire est $\lfloor \lg n \rfloor$, les trois opérations d'une file de priorité peuvent se faire chacune en un temps $O(\lg n)$. Pour INSÉRER, il suffit d'ajouter une nouvelle feuille au premier emplacement disponible. Ensuite, tant que la clé du nœud est supérieure à celle du nœud parent, nous échangeons le contenu[1] du nœud avec celui de son parent et remontons d'un niveau vers la racine. Autrement dit, nous déplaçons le contenu vers la racine jusqu'à ce la propriété de tas soit respectée. Puisque le chemin vers la racine comprend au plus $\lfloor \lg n \rfloor$ arêtes, nous effectuons au plus $\lfloor \lg n \rfloor - 1$ échanges. INSÉRER prend donc un temps $O(\lg n)$. Pour DIMINUER-CLÉ, nous employons la même idée : décrémenter la clé et remonter le contenu vers la racine jusqu'à ce que la propriété de tas soit satisfaite, encore une fois en un temps $O(\lg n)$. Pour EXTRAIRE-MIN, nous enregistrons le contenu de la

1. Le contenu d'un nœud comprend la clé et les autres informations liées à cette clé, comme le sommet associé au nœud.

Copyright © 2013 Dunod.

racine à retourner à l'appelant. Ensuite, nous prenons la dernière feuille (le nœud de numéro le plus élevé) et plaçons son contenu à la racine. Ensuite, nous descendons le contenu de la racine, en l'échangeant avec celui de l'enfant dont la clé est inférieure, jusqu'à ce que la propriété de tas soit respectée. Pour finir, nous retournons le contenu enregistré. De nouveau, puisque le chemin de la racine à une feuille comprend au plus $\lfloor \lg n \rfloor$ arêtes, nous effectuons au plus $\lfloor \lg n \rfloor - 1$ échanges et EXTRAIRE-MIN demande un temps $O(\lg n)$.

Lorsque l'algorithme de Dijkstra utilise une file de priorité implémentée sous forme de tas binaire, l'insertion des sommets se fait en un temps $O(n \lg n)$, les opérations EXTRAIRE-MIN en un temps $O(n \lg n)$ et les opérations DIMINUER-CLÉ en un temps $O(m \lg n)$. (En réalité, l'insertion des $n$ sommets peut se faire en un temps $\Theta(n)$ seulement, car seul le sommet source $s$ a initialement une valeur *pluscourt* égale à 0 et tous les autres sommets ont une valeur *pluscourt* égale à $\infty$.) Lorsque le graphe est **peu dense** (le nombre $m$ d'arêtes est largement inférieur à $n^2$), l'implémentation de la file de priorité sous forme de tas binaire est plus efficace que celle fondée sur un simple tableau. Les graphes qui modélisent des réseaux routiers sont peu denses, car d'une intersection part en moyenne quatre routes et $m$ est alors environ égal à $4n$. *A contrario*, lorsque le graphe est **dense** ($m$ est proche de $n^2$ et le graphe comprend de nombreuses arêtes), le temps $O(m \lg n)$ que l'algorithme de Dijkstra passe dans les appels à DIMINUER-CLÉ peut le rendre plus lent qu'avec une file de priorité implémentée par un simple tableau.

Sachez également que les tas binaires peuvent être triés en un temps $O(n \lg n)$.

*Procédure* TRI-PAR-TAS($A$,$n$)
*Entrées* :
- $A$ : un tableau.
- $n$ : le nombre d'éléments de $A$ à trier.

*Sortie* : Un tableau $B$ qui contient les éléments de $A$ triés.

1. Créer un tas binaire $Q$ à partir des éléments de $A$.
2. Créer un tableau $B[1..n]$.
3. Pour $i = 1$ à $n$ :
   A. Appeler EXTRAIRE-MIN($Q$) et affecter la valeur retournée à $B[i]$.
4. Retourner le tableau $B$.

L'étape 1 se charge de la conversion du tableau d'entrée en un tas binaire, une opération qui peut se réaliser de deux manières. La première consiste à partir d'un tas binaire vide et à insérer chaque élément

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

du tableau ; elle demande un temps $O(n \lg n)$. La seconde consiste à construire le tas binaire directement dans le tableau, en travaillant du bas vers le haut ; elle prend un temps $O(n)$ seulement. Il est également possible d'effectuer un tri en utilisant un tas sur place, ce qui évite le tableau $B$ supplémentaire.

### Implémentation avec un tas de Fibonacci

L'implémentation de la file de priorité peut également se faire à l'aide d'une structure de données complexe appelée *tas de Fibonacci*. Dans ce cas, les $n$ appels à INSÉRER et à EXTRAIRE-MIN prennent un temps total $O(n \lg n)$ et les $m$ appels à DIMINUER-CLÉ un temps total $\Theta(m)$. L'algorithme de Dijkstra s'exécute alors en un temps $O(n \lg n + m)$ seulement. Dans la pratique, les tas de Fibonacci sont rarement employés, cela pour deux raisons. La première est qu'une opération individuelle peut prendre beaucoup plus de temps que la moyenne, même si globalement les opérations se font dans les temps indiqués précédemment. La seconde est que ces tas sont un tantinet complexes et que les facteurs constants masqués par la notation asymptotique ne sont pas aussi bons que pour les tas binaires.

## ALGORITHME DE BELLMAN-FORD

Si le poids de certains arcs sont négatifs, l'algorithme de Dijkstra risque de retourner des résultats incorrects. L'algorithme de Bellman-Ford[1] prend en charge les poids négatifs et peut nous servir à détecter et à identifier un circuit de poids négatif.

Cet algorithme est remarquablement simple. Après que *pluscourt* et *pred* ont été initialisés, il suffit de relâcher les $m$ arcs à $n-1$ reprises. La procédure est donnée ci-après et la figure illustre le fonctionnement de l'algorithme sur un petit graphe. Le sommet source est $s$, la valeur de *pluscourt* est indiquée à l'intérieur des sommets et les arcs grisés révèlent les valeurs de *pred* : si l'arc $(u,v)$ est grisé alors $pred[v] = u$. Dans cet exemple, nous supposons que chaque passe sur tous les arcs les relâche dans l'ordre figé $(t,x)$, $(t,y)$, $(t,z)$, $(x,t)$, $(y,x)$, $(y,z)$, $(z,x)$, $(z,s)$, $(s,t)$, $(s,y)$. La partie (a) montre la configuration juste avant la première passe. Les parties (b) à (e) la montrent après chaque passe successive. Les valeurs de *pluscourt* et de *pred* en partie (e) sont les valeurs finales.

1. Il se fonde sur des algorithms distincts conçus par Richard Bellman en 1958 et par Lester Ford en 1962.

Copyright © 2013 Dunod.

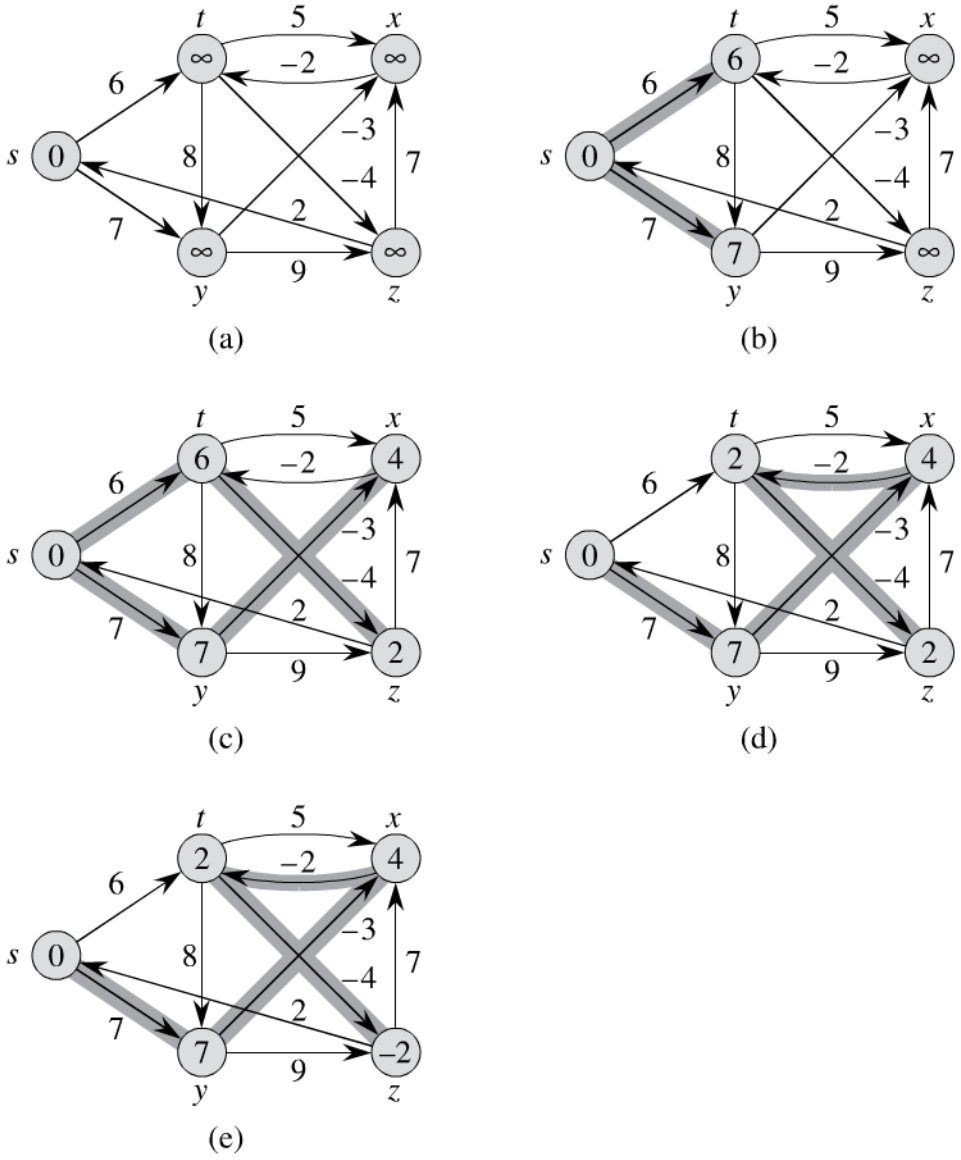
*Procédure* BELLMAN-FORD$(G,s)$

*Entrées :*

- $G$ : un graphe orienté qui contient un ensemble $V$ de $n$ sommets et un ensemble $E$ de $m$ arcs avec des poids quelconques.
- $s$ : un sommet source dans $V$.

*Résultat :* Identique à celui de DIJKSTRA (page 99).

1. Fixer *pluscourt*$[v]$ à $\infty$ pour chaque sommet $v$, excepté $s$, fixer *pluscourt*$[s]$ à 0 et fixer *pred*$[v]$ à NULL pour tout sommet $v$.
2. Pour $i = 1$ à $n - 1$ :
   A. Pour chaque arc $(u,v)$ de $E$ :
      i. Appeler RELÂCHEMENT$(u,v)$.




Comment un algorithme aussi simple peut-il donner une réponse exacte ? Prenons un plus court chemin à partir de la source $s$ vers n'importe quel

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

sommet $v$. À la page 90, nous avons expliqué que si nous relâchons les arcs, dans l'ordre, le long d'un plus court chemin de $s$ à $v$, alors $pluscourt[v]$ et $pred[v]$ sont correctes. À présent, si les circuits de poids négatif sont interdits, il existe toujours un plus court chemin de $s$ à $v$ qui ne contient aucun circuit. Pourquoi ? Supposons qu'un plus court chemin de $s$ à $v$ comprenne un circuit. Puisque le circuit doit avoir un poids positif, nous pouvons le couper et arriver à un chemin de $s$ à $v$ dont le poids n'est pas plus élevé que le chemin qui comprend le circuit. Chaque chemin acyclique doit contenir au plus $n-1$ arcs, car un chemin qui comprendrait $n$ arcs doit passer deux fois par un sommet, ce qui crée un circuit. Par conséquent, s'il existe un plus court chemin de $s$ à $v$, alors il doit contenir au plus $n-1$ arcs. La première fois que l'étape 2A relâche tous les arcs, elle doit relâcher le premier arc sur le plus court chemin. La deuxième fois qu'elle relâche tous les arcs, elle doit relâcher le deuxième arc sur le plus court chemin, et ainsi de suite. Après la $(n-1)$-ème fois, tous les arcs sur le plus court chemin ont été relâchés, dans l'ordre, et $pluscourt[v]$ et $pred[v]$ sont correctes. Plutôt habile !

Supposons à présent que le graphe contienne un circuit de poids négatif et que nous lui ayons déjà appliqué la procédure Bellman-Ford. Nous pouvons boucler dans un circuit de poids négatif, en obtenant un chemin de poids inférieur à chaque tour. Cela signifie qu'il existe dans le circuit au moins un arc $(u,v)$ pour lequel $pluscourt[v]$ va diminuer s'il est de nouveau relâché, même si cet arc a déjà été relâché à $n-1$ reprises.

Voici donc comment rechercher un circuit de poids négatif, s'il existe, après avoir exécuté la procédure Bellman-Ford. Nous parcourons à nouveau tous les arcs et, si nous trouvons un arc $(u,v)$ pour lequel $pluscourt[u] + poids(u,v) < pluscourt[v]$, alors nous savons que ce sommet $v$ est dans un circuit de poids négatif ou qu'il peut être atteint à partir d'un tel circuit. Pour rechercher un sommet sur un circuit de poids négatif, nous examinons à reculons les valeurs de *pred* à partir de $v$, en conservant une trace des sommets visités, jusqu'à ce que nous atteignions un sommet $x$ que nous avons déjà rencontré. Ensuite, nous pouvons examiner à reculons les valeurs de *pred* à partir de $x$ jusqu'à ce que nous revenions à $x$. Tous les sommets rencontrés lors de ce parcours, avec $x$, constituent un circuit de poids négatif. La procédure Chercher-Circuit-Poids-Négatif donnée ci-après montre comment déterminer si un graphe comprend un circuit de poids négatif et comment en construire un si c'est le cas.

*Procédure* Chercher-Circuit-Poids-Négatif($G$)

*Entrée* : $G$ : un graphe orienté qui contient un ensemble $V$ de $n$ sommets et un ensemble $E$ de $m$ arcs avec des poids quelconques sur lequel la procédure Bellman-Ford a déjà été exécutée.

Copyright © 2013 Dunod.

*Sortie* : Une liste de sommets compris dans un circuit de poids négatif, dans l'ordre, ou une liste vide si le graphe ne comprend aucun circuit de poids négatif.

1. Parcourir tous les arcs à la recherche d'un arc $(u,v)$ tel que $pluscourt[u] + poids(u,v) < pluscourt[v]$.
2. Si un tel arc n'existe pas, retourner une liste vide.
3. Sinon (il existe un arc $(u,v)$ pour lequel $pluscourt[u] + poids(u,v) < pluscourt[v]$), faire la chose suivante :
   A. Soit *visités* un tableau avec un élément pour chaque sommet. Fixer tous les éléments de *visités* à FAUX.
   B. Fixer $x$ à $v$.
   C. Tant que *visités*[$x$] est FAUX, faire la chose suivante :
      i. Fixer *visités*[$x$] à VRAI.
      ii. Fixer $x$ à *pred*[$x$].
   D. À ce stade, nous savons que $x$ est un sommet sur un circuit de poids négatif. Fixer $v$ à *pred*[x].
   E. Créer une liste de sommets *circuit* qui comprend initialement uniquement $x$.
   F. Tant que $v$ n'est pas $x$, faire la chose suivante :
      i. Insérer le sommet $v$ au début de *circuit*.
      ii. Fixer $v$ à *pred*[$v$].
   G. Retourner *circuit*.

L'analyse du temps d'exécution de l'algorithme Bellman-Ford ne pose pas de difficultés. La boucle de l'étape 2 réalise $n - 1$ itérations et, à chaque tour, la boucle de l'étape 2A itère à $m$ reprises, une fois par arc. Par conséquent, le temps d'exécution total est $\Theta(nm)$. Pour rechercher s'il existe un circuit de poids négatif, nous relâchons chaque arc une fois de plus jusqu'à ce que cette opération modifie une valeur de *pluscourt* ou que tous les arcs aient été relâchés. Cela se fait en un temps $O(m)$. S'il existe un circuit de poids négatif, il peut contenir au plus $n$ arcs et le temps d'analyse est donc $O(n)$.

Au début de ce chapitre, j'ai promis de montrer comment des circuits de poids négatif interviennent dans l'arbitrage d'opportunités d'échanges monétaires. Les taux de change des devises fluctuent rapidement. Imaginons qu'à un moment donné, ils soient les suivants :

- 1 dollar US vaut 0,7292 euro ;
- 1 euro vaut 105,374 yens japonais ;
- 1 yen japonais vaut 0,3931 rouble russe ;
- 1 rouble russe vaut 0,0341 dollar US.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Avec 1 dollar US, nous pouvons donc acheter 0,7292 euro, avec lesquels nous pouvons acheter 76,838 7 yens (car 0,729 2 · 105,374 = 76,838 7, avec quatre décimales), avec lesquels nous pouvons acheter 30,205 3 roubles (car 76,838 7 · 0,393 1 = 30,205 3, avec quatre décimales), avec lesquels nous pouvons acheter 1,03 dollars (car 30,205 3 · 0,034 1 = 1,030 0, avec quatre décimales). Si nous pouvons réaliser les quatre transactions avant que les taux de change n'évoluent, nous obtenons un profit de 3 % sur notre investissement de 1 dollar. En partant avec 1 million de dollars, nous avons un bénéfice de 30 000 dollars sans rien faire !

Un tel scénario constitue une **opportunité d'arbitrage**. Voici comment nous pouvons trouver une opportunité d'arbitrage en recherchant un circuit de poids négatif. Supposons que nous examinions $n$ devises ($c_1$, $c_2$, $c_3$,..., $c_n$) et que nous disposions de tous les taux de change entre chaque couple de monnaie. Supposons qu'avec une unité de la devise $c_i$ nous puissions acheter $r_{ij}$ unités de la devise $c_j$ ; $r_{ij}$ est le taux de change entre les devises $c_i$ et $c_j$. Dans ce cas, $i$ et $j$ sont dans la plage 1 à $n$. (On peut supposer que $r_{ii} = 1$ pour toute devise $c_i$.)

Une opportunité d'arbitrage va correspondre à une suite de $k$ devises, $<c_{j1}, c_{j2}, c_{j3},..., c_{jk}>$ telle que la multiplication des taux de change donne un résultat strictement supérieur à 1 :

$$r_{j1,j2} \cdot r_{j2,j3} \cdots r_{jk-1,jk} \cdot r_{jk,j1} > 1$$

Prenons à présent le logarithme de chaque côté. Puisque la base utilisée n'a pas d'importance, jouons à l'informaticien et choisissons la base 2. Puisque le logarithme d'un produit est la somme des logarithmes individuels – $\lg(x \cdot y) = \lg x + \lg y$ –, nous recherchons un cas où :

$$\lg r_{j1,j2} + \lg r_{j2,j3} + \ldots + \lg r_{jk-1,jk} + \lg r_{jk,j1} > 0$$

En inversant les deux côtés de cette inégalité, nous obtenons :

$$\left(-\lg r_{j1,j2}\right) + \left(-\lg r_{j2,j3}\right) + \ldots + \left(-\lg r_{jk-1,jk}\right) + \left(-\lg r_{jk,j1}\right) < 0$$

Cela correspond à un circuit avec des arcs dont les poids sont les inverses des logarithmes des taux de change.

Pour rechercher une éventuelle opportunité d'arbitrage, nous construisons un graphe orienté avec un sommet $v_i$ pour chaque devise $c_i$. Pour chaque couple de devises $c_i$ et $c_j$, nous créons des arcs $(v_i, v_j)$ et $(v_j, v_i)$, avec, respectivement, des poids $-\lg r_{ij}$ et $-\lg r_{ji}$. Nous ajoutons un nouveau sommet $s$ avec un arc $(s, v_i)$ de poids 0 vers chacun des sommets $v_1$ à $v_n$. Nous exécutons l'algorithme de Bellman-Ford sur ce graphe en prenant le sommet source $s$ et nous utilisons le résultat pour déterminer s'il contient un circuit de poids négatif. Dans l'affirmative, les sommets de ce circuit correspondent aux devises qui présentent une opportunité d'arbitrage. Le nombre total d'arcs $m$ est $n + n(n-1) = n^2$. L'algorithme de Bellman-Ford s'exécute donc en un temps $O(n^3)$. Il faut également un temps $O(n^2)$ pour

Copyright © 2013 Dunod.

déterminer l'existence d'un circuit de poids négatif, ainsi qu'un autre temps $O(n)$ pour l'identifier. Bien que la durée $O(n^3)$ puisse sembler longue, dans la pratique elle n'est pas si mauvaise car les facteurs constants dans les circuits sont faibles. J'ai implémenté le programme d'arbitrage sur mon MacBook Pro à 2,4 GHz et l'est exécuté avec 182 devises (toutes celles du monde entier). Après que j'ai chargé les taux de change (des valeurs au hasard), l'exécution du programme a mis environ 0,02 seconde.

## Algorithme de Floyd-Warshall

Supposons à présent que nous voulions trouver un plus court chemin entre chaque sommet. Il s'agit du problème des **plus courts chemins pour tout couple de sommets**.

L'exemple classique, auquel de nombreux auteurs font référence, est celui d'une table des distances entre plusieurs villes, comme on peut en trouver dans un atlas routier. Il suffit de rechercher la ligne de la première ville, la colonne de la seconde, et la distance qui les sépare se trouve à l'intersection de cette ligne et de cette colonne.

Cet exemple pose toutefois un problème : il ne traite *pas tous les couples*. Dans le cas contraire, la table comprendrait une ligne et une colonne non pas pour chaque ville mais pour chaque intersection. Leur nombre pour la France se chiffrerait en millions. Pour créer la table d'un atlas routier, la solution consiste à rechercher le plus court chemin à origine unique à partir de chaque ville, puis à ne retenir qu'un sous-ensemble des résultats, c'est-à-dire les plus courts chemins vers les autres villes non les intersections.

Quelle serait une application légitime des plus courts chemins pour tous les couples de sommets ? Rechercher le **diamètre** d'un réseau, c'est-à-dire le plus long des plus courts chemins. Par exemple, supposons qu'un graphe orienté représente un réseau de communication et que le poids d'un arc corresponde au temps qu'il faut à un message pour traverser un lien de communication. Le diamètre permet de connaître la plus longue durée de transmission d'un message au sein du réseau.

Nous pouvons évidemment calculer les plus courts chemins pour tout couple de sommets en déterminant le plus court chemin à origine unique pour chaque sommet. Si tous les poids des arcs sont positifs, nous pouvons appliquer l'algorithme de Dijkstra à partir de chacun des $n$ sommets, chaque appel prenant un temps $O(m \lg n)$ avec un tas binaire et un temps $O(n \lg n + m)$ avec un tas de Fibonacci. Le temps d'exécution total est donc $O(n\ m \lg n)$ ou $O(n^2 \lg n + nm)$. Si le graphe est peu dense, cette approche convient. En revanche, si le graphe est dense ($m$ est proche de $n^2$), alors $O(n \lg n + m)$ devient $O(n^3 \lg n)$. Même dans le cas d'un graphe dense et d'un tas de Fibonacci, $O(n^2 \lg n + nm)$ est $O(n^3)$, et le facteur constant

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

introduit par le tas de Fibonacci peut être significatif. Bien entendu, si le graphe risque d'inclure des arcs de poids négatif, l'algorithme de Dijkstra est inutilisable et l'exécution de l'algorithme de Bellman-Ford pour chacun des $n$ sommets d'un graphe dense conduit à un temps d'exécution de $\Theta(n^2m)$, c'est-à-dire $\Theta(n^4)$.

En utilisant de Floyd-Warshall[1], nous pouvons résoudre le problème pour tous les couples en un temps $\Theta(n^3)$, quelles que soient la densité du graphe et l'existence de circuits de poids négatif. De plus, le facteur constant masqué par la notation $\Theta$ est faible. Par ailleurs, l'algorithme de Floyd-Warshall est une illustration d'une technique algorithmique intelligente appelée « programmation dynamique ».

L'algorithme de Floyd-Warshall se fonde sur une propriété évidente des plus courts chemins. Supposons que nous allions de Bordeaux à Paris en suivant l'itinéraire le plus court et que celui-ci passe par Nantes puis par Rennes. Alors, la partie du plus court chemin de Bordeaux à Paris qui va de Nantes à Rennes doit être elle-même un plus court chemin de Nantes à Rennes. Pourquoi cela ? Parce que s'il existe un plus court chemin de Nantes à Rennes, alors nous l'avons emprunté dans le plus court chemin de Bordeaux à Paris ! Appliquons ce principe aux graphes orientés :

> Si un plus court chemin, appelé $p$, entre le sommet $u$ et le sommet $v$ va du sommet $u$ au sommet $x$, puis au sommet $y$ et enfin au sommet $v$, alors la portion de $p$ qui se trouve entre $x$ et $y$ est elle-même un plus court chemin entre $x$ et $y$. Autrement dit, *tout sous-chemin d'un plus court chemin est lui-même un plus court chemin.*

L'algorithme de Floyd-Warshall gère les poids des chemins et les prédécesseurs des sommets dans des tableaux indicés non pas en une, mais en trois dimensions. Un tableau à une dimension peut être considéré comme une table (voir à la page 12). Un tableau à deux dimensions peut être vu comme une matrice, par exemple la matrice d'adjacence à la page 83 ; il faut deux indices (ligne et colonne) pour identifier une entrée. Un tableau à deux dimensions peut également être vu comme un tableau à une dimension dont chaque entrée est elle-même un tableau à une dimension. Un tableau à trois dimensions est comparable à un tableau à une dimension dont chaque entrée est un tableau à deux dimensions ; pour identifier une entrée, nous avons besoin d'un indice sur chacune des trois dimensions. Pour écrire un indice d'un tableau à plusieurs dimensions, nous séparons chaque dimension par une virgule.

Dans l'algorithme de Floyd-Warshall, nous supposons que les sommets sont numérotés de 1 à $n$. Ces numéros de sommets sont importants car l'algorithme se fonde sur la définition suivante :

1. Ainsi nommé d'après Robert Floyd et Stephen Warshall.

Copyright © 2013 Dunod.

> *pluscourt*$[u,v,x]$ est le poids d'un plus court chemin qui va du sommet $u$ au sommet $v$ et dans lequel chaque sommet intermédiaire – un sommet du chemin autre que $u$ et $v$ – est numéroté de 1 à $x$.

(Voyez $u$, $v$ et $x$ comme des entiers dans la plage 1 à $n$ qui représentent des sommets.) Cette définition n'impose pas que les sommets intermédiaires soient constitués de tous les sommets x numéroté de 1 à $x$. Elle exige simplement que chaque sommet intermédiaire, quel que soit leur nombre, ait un numéro inférieur ou égal à $x$. Puisque tous les sommets ont un numéro inférieur ou égal à $n$, nous devons avoir le cas où *pluscourt*$[u,v,n]$ est égal à $sp(u,v)$, le poids d'un plus court chemin de $u$ à $v$.

Prenons deux sommets $u$ et $v$, et choisissons un nombre $x$ entre 1 et $n$. Considérons tous les chemins de $u$ à $v$ dans lesquels tous les sommets intermédiaires ont un numéro inférieur ou égal à $x$. Parmi tous ces chemins, supposons que le chemin $p$ ait un poids minimal. Soit le chemin $p$ comprend un sommet $x$, soit il ne le comprend pas. Et nous savons que, hormis $u$ ou $v$, il ne comprend aucun sommet de numéro supérieur à $x$. Il existe deux possibilités :

- **Première possibilité :** $x$ n'est pas un sommet intermédiaire dans le chemin $p$. Alors, tous les sommets intermédiaires du chemin $p$ ont des numéros inférieurs ou égaux à $x-1$. Cela signifie que le poids d'un plus court chemin de $u$ à $v$ avec tous les sommets intermédiaires de numéro inférieur à $x$ est identique au poids d'un plus court chemin de $u$ à $v$ avec tous les sommets intermédiaires de numéro inférieur ou égal à $x-1$. Autrement dit, *pluscourt*$[u,v,x]$ est égal à *pluscourt*$[u,v,x-1]$.
- **Seconde possibilité :** $x$ est un sommet intermédiaire dans le chemin $p$. Puisque tout sous-chemin d'un plus court chemin est lui-même un plus court chemin, la portion du chemin $p$ qui va de $u$ à $x$ est un plus court chemin de $u$ à $x$. De même, la portion de $p$ qui va de $x$ à $v$ est un plus court chemin de $x$ à $v$. Puisqu'un sommet $x$ est une extrémité de chacun de ces sous-chemins, il n'est pas un sommet intermédiaire dans aucun d'eux et les sommets intermédiaires de chacun de ces sous-chemins ont tous des numéros inférieurs ou égaux à $x-1$. Par conséquent, le poids d'un sous-chemin de $u$ à $v$ avec tous les sommets intermédiaires ayant des numéros inférieurs ou égaux à $x$ est la somme des poids de deux plus courts chemins : l'un qui va de $u$ à $x$ avec tous les sommets intermédiaires de numéro inférieur ou égal à $x-1$, et l'autre allant de $x$ à $v$, également avec tous les sommets intermédiaires de numéro inférieur ou égal à $x-1$. Autrement dit, *pluscourt*$[u,v,x]$ est égal à *pluscourt*$[u,x,x-1]$ + *pluscourt*$[x,v,x-1]$.

Puisque $x$ est ou n'est pas un sommet intermédiaire dans un plus court chemin de $u$ à $v$, nous pouvons en conclure que *pluscourt*$[u,v,x]$ est le minimum entre *pluscourt*$[u,x,x-1]$ + *pluscourt*$[x,v,x-1]$ et *pluscourt*$[u,v,x-1]$.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Dans l'algorithme de Floyd-Warshall, la meilleure façon de représenter un graphe consiste à utiliser une variante de la matrice d'adjacence illustrée page 82. Au lieu que chaque élément de la matrice se limite aux valeurs 0 et 1, l'entrée de l'arc $(u,v)$ contient le poids de celui-ci, la valeur $\infty$ indiquant que l'arc n'existe pas. Puisque $pluscourt[u,v,0]$ donne le poids d'un plus court chemin de $u$ à $v$, avec tous les sommets intermédiaires de numéro inférieur ou égal à 0, un tel chemin ne comprend aucun sommet intermédiaire. Autrement dit, il est constitué d'un seul arc et la matrice correspond exactement à ce que nous voulons pour $pluscourt[u,v,0]$.

À partir des valeurs $pluscourt[u,v,0]$ (les poids des arcs), l'algorithme de Floyd-Warshall commence par calculer les valeurs $pluscourt[u,v,x]$ pour tous les couples de sommets $u$ et $v$ avec $x$ fixé à 1. Ensuite, il calcule les valeurs $pluscourt[u,v,x]$ pour tous les couples de sommets $u$ et $v$ avec $x$ fixé 2. Il continue avec $x$ fixé à 3, et ainsi de suite jusqu'à $n$.

Comment se passe la gestion des prédécesseurs ? Définissons $pred[u,v,x]$ de manière analogue à la définition de $pluscourt[u,v,x]$ : il s'agit du prédécesseur d'un sommet $v$ sur un plus court chemin partant du sommet $u$ dans lequel tous les sommets intermédiaires ont un numéro inférieur ou égal à $x$. Nous pouvons actualiser les valeurs $pred[u,v,x]$ pendant le calcul des valeurs $pluscourt[u,v,x]$, de la manière suivante. Si $pluscourt[u,v,x]$ est égal à $pluscourt[u,v,x-1]$, alors le plus court chemin que nous avons trouvé entre $u$ et $v$ avec tous les sommets intermédiaires de numéro inférieur ou égal à $x$ est identique à celui avec tous les sommets intermédiaires de numéro inférieur ou égal à $x-1$. Le prédécesseur du sommet $v$ doit être le même dans les deux chemins et nous pouvons fixer $pred[u,v,x]$ à la même valeur que $pred[u,v,x-1]$. Qu'en est-il lorsque $pluscourt[u,v,x]$ est inférieur à $pluscourt[u,v,x-1]$ ? Cela se produit lorsque nous trouvons un chemin de $u$ à $v$ qui possède un sommet intermédiaire $x$ et un poids inférieur à celui du plus court chemin de $u$ à $v$ avec tous les sommets intermédiaires de numéro inférieur ou égal à $x-1$. Puisque $x$ doit être un sommet intermédiaire sur ce nouveau plus court chemin trouvé, le prédécesseur de $v$ sur le chemin partant de $u$ doit être le même que le prédécesseur de $v$ sur le chemin partant de $x$. Dans ce cas, nous fixons $pred[u,v,x]$ à la valeur de $pred[u,v,x-1]$.

Nous disposons à présent de tous les éléments pour assembler l'algorithme de Floyd-Warshall. Voici la procédure correspondante :

*Procédure* FLOYD-WARSHALL($G$)

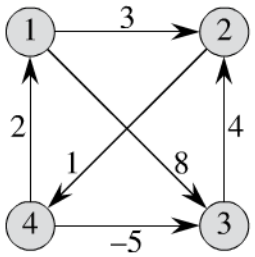
*Entrée :* $G$ : un graphe représenté par une matrice d'adjacence pondérée $W$, avec $n$ lignes et $n$ colonnes (une ligne et une colonne pour chaque sommet). L'entrée à l'intersection de la ligne $u$ et de la colonne $v$ s'écrit $w_{uv}$ ; elle donne le poids de l'arc $(u,v)$ s'il existe dans $G$, sinon $\infty$.

Copyright © 2013 Dunod.

*Sortie :* pour chaque couple de sommets $u$ et $v$, la valeur de $pluscourt[u,v,n]$ donne le poids d'un plus court chemin qui va de $u$ à $v$, et $pred[u,v,n]$ est le sommet qui précède $v$ sur un plus court chemin partant de $u$.

1. Créer les tableaux *pluscourt* et *pred* de taille $n \times n \times (n+1)$.
2. Pour chaque $u$ et $v$ allant de 1 à $n$ :
   A. Fixer $pluscourt[u,v,0]$ à $w_{uv}$.
   B. Si $(u,v)$ est un arc de $G$, alors fixer $pred[u,v,0]$ à $u$. Sinon fixer $pred[u,v,0]$ à NULL.
3. Pour $x = 1$ à $n$ :
   A. Pour $u = 1$ à $n$ :
      i. Pour $v = 1$ à $n$ :
         a. Si $pluscourt[u,v,x] < pluscourt[u,x,x-1] + pluscourt[x,v,x-1]$, alors fixer $pluscourt[u,v,x]$ à $pluscourt[u,x,x-1] + pluscourt[x,v,x-1]$ et fixer $pred[u,v,x]$ à $pred[x,v,x-1]$.
         b. Sinon fixer $pluscourt[u,v,x]$ à $pluscourt[u,v,x-1]$ et fixer $pred[u,v,x]$ à $pred[u,v,x-1]$.
4. Retourner les tableaux *pluscourt* et *pred*.

Soit le graphe suivant :




Sa matrice d'adjacence $W$, qui contient les poids des arcs, est la suivante :

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix}$$

Elle donne également les valeurs $pluscourt[u,v,0]$[1] (les poids des chemins ayant au plus un arc). Par exemple, $pluscourt[2,4,0]$ vaut 1, car nous pouvons aller directement du sommet 2 au sommet 4, sans passer par des sommets intermédiaires, en empruntant l'arc (2,4) de poids 1. De manière comparable, $pluscourt[4,3,0]$ vaut –5. Voici une matrice qui donne les valeurs $pred[u,v,0]$ :

1. Puisqu'un tableau à trois dimensions est un tableau à une dimension dont les entrées sont des tableaux à deux dimensions, pour une valeur de $x$, nous pouvons traiter $pluscourt[u,v,x]$ comme un tableau à deux dimensions.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

$$\begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}$$

Par exemple, $pred[2,4,0]$ vaut 2 car le prédécesseur du sommet 4 est le sommet 2 en suivant l'arc (2,4) de poids 1, et $pred[2,3,0]$ est NULL car il n'existe aucun arc (2,3).

Après avoir exécuté la boucle de l'étape 3 pour $x = 1$ (afin d'examiner les chemins qui peuvent comprendre le sommet intermédiaire 1), les valeurs $pluscourt[u,v,1]$ et $pred[u,v,1]$ sont :

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}.$$

Après l'exécution de la boucle pour $x = 2$, les valeurs de $pluscourt[u,v,2]$ et de $pred[u,v,2]$ sont :

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}.$$

Les voici pour $x = 3$ :

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

Copyright © 2013 Dunod.

Les valeurs finales de $pluscourt[u,v,4]$ et de $pred[u,v,4]$ sont obtenues après l'exécution de la boucle pour $x = 4$ :

$$\begin{pmatrix} 0 & 3 & -1 & 4 \\ 3 & 0 & -4 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ et } \begin{pmatrix} \text{NULL} & 1 & 4 & 2 \\ 4 & \text{NULL} & 4 & 2 \\ 4 & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

Nous pouvons voir, par exemple, que le plus court chemin du sommet 1 au sommet 3 a un poids égal à -1. Ce chemin passe par les sommets 1, 2, 4 et 3, dans cet ordre, ce que nous pouvons déterminer en examinant les valeurs de $pred$ : $pred[1,3,4]$ vaut 4, $pred[1,4,4]$ vaut 2, et $pred[1,2,4]$ vaut 1.

J'ai prétendu que l'algorithme de Floyd-Warshall s'exécute en un temps $\Theta(n^3)$, et cela est facile à vérifier. Nous avons des boucles imbriquées à trois niveaux et chacune réalise $n$ itérations. Dans chaque itération de la boucle de l'étape 3, la boucle de l'étape 3A itère à $n$ reprises. De même, dans chaque itération de la boucle de l'étape 3A, la boucle de l'étape 3Ai itère à $n$ reprises. Puisque la boucle externe de l'étape 3 itère également à $n$ reprises, la boucle interne (étape 3Ai) effectue $n^3$ itérations. Chaque itération de cette boucle se fait en un temps constant. L'algorithme prend donc un temps $\Theta(n^3)$.

Il semblerait également que l'occupation mémoire de cet algorithme soit $\Theta(n^3)$. En effet, il crée deux tableaux $n \times n \times (n+1)$. Puisque chaque entrée de tableau utilise une quantité de mémoire constante, ces tableaux occupent un espace mémoire $\Theta(n^3)$. Mais nous pouvons faire en sorte que l'espace mémoire occupé soit uniquement $\Theta(n^2)$. Pour cela, il suffit de créer *pluscourt* et *pred* sous forme de tableaux $n \times n$ et d'omettre le troisième indice dans toutes les références à *pluscourt* et à *pred*. Bien que les étapes 3Aia et 3Aib actualisent les mêmes valeurs $pluscourt[u,v]$ et $pred[u,v]$, ces tableaux finiront par avoir les valeurs correctes !

J'ai mentionné précédemment que l'algorithme de Floyd-Warshall illustre une technique de **programmation dynamique**. Cette technique s'applique uniquement :

1. lorsque nous tentons de trouver une solution optimale à un problème,
2. nous pouvons décomposer une instance du problème en instances d'un ou plusieurs sous-problèmes,
3. nous utilisons les solutions des sous-problèmes pour résoudre le problème d'origine, et
4. si nous utilisons une solution d'un sous-problème dans une solution optimale pour le problème origine, alors la solution du sous-problème utilisée doit être optimale pour le sous-problème.

Nous pouvons réunir ces conditions sous le nom **sous-structure optimale**. De façon plus succincte, cela signifie qu'une solution optimale à un problème comprend des solutions optimales à des sous-problèmes. En programmation dynamique, nous avons la notion de « taille » d'un sous-problème et nous résolvons souvent les sous-problèmes par ordre croissant de taille. Les plus petits sous-problèmes sont donc résolus en premier puis, avec les solutions optimales trouvées, nous essayons de résoudre des sous-problèmes plus grands de façon optimale.

Cette description de la programmation dynamique semble plutôt abstraite. Voyons donc comment l'algorithme de Floyd-Warshall l'exploite. Voici notre définition d'un sous-problème :

> Calculer $pluscourt[u,v,x]$, qui est le poids d'un plus court chemin partant du sommet $u$ et allant au sommet $v$ et dans lequel chaque sommet intermédiaire est numéroté de 1 à $x$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans ce cas, la « taille » d'un sous-problème est le sommet qui possède le numéro le plus élevé et que nous acceptons comme sommet intermédiaire dans un plus court chemin : autrement dit, il s'agit de la valeur de $x$. Une sous-structure optimale existe en raison de la propriété suivante :

> Considérons un plus court chemin $p$ depuis le sommet $u$ vers le sommet $v$ et nommons $x$ le sommet intermédiaire de numéro le plus élevé dans ce chemin. Alors, la portion de $p$ qui va de $u$ à $x$ est le plus court chemin qui va de $u$ à $x$ avec tous les sommets intermédiaires numérotés de 1 à $x-1$, et la portion de $p$ qui va de $x$ à $v$ est un plus court chemin qui va de $x$ à $v$ avec tous les sommets intermédiaires numérotés de 1 à $x-1$.

Nous résolvons le problème du calcul de $pluscourt[u,v,x]$ en commençant par déterminer $pluscourt[u,v,x-1]$, $pluscourt[u,x,x-1]$ et $pluscourt[x,v,x-1]$, puis en utilisant le plus petit terme entre $pluscourt[u,v,x-1]$ et $pluscourt[u,x,x-1]+pluscourt[x,v,x-1]$. Puisque nous avons calculé toutes les valeurs de *pluscourt* pour lesquelles le troisième indice est $x-1$ avant d'essayer de calculer toute valeur de *pluscourt* pour lesquelles le troisième indice est $x$, nous avons toutes les informations dont nous avons besoin lorsque nous calculons $pluscourt[u,v,x]$.

En programmation dynamique, une pratique courante consiste à stocker les solutions optimales des sous-problèmes ($pluscourt[u,v,x-1]$, $pluscourt[u,x,x-1]$ et $pluscourt[x,v,x-1]$) dans une table et à les rechercher lors du calcul d'une solution optimale pour le problème d'origine ($pluscourt[u,v,x]$). Cette approche est dite « ascendante » (*bottom up*), car elle opère sur des sous-problèmes plus petits pour passer ensuite aux sous-problèmes plus grands. Une autre approche consiste à résoudre les sous-problèmes de manière « descendante » (*top down*), en commençant par les plus grands pour aller vers les plus petits, à nouveau en stockant les résultats de chaque sous-problème dans une table.

La programmation dynamique s'applique à une grande diversité de problèmes d'optimisation, dont seuls certains concernent les graphes. Nous y reviendrons au Chapitre 7, lorsque nous rechercherons la plus longue sous-séquence commune à deux chaînes de caractères.

Copyright © 2013 Dunod.

## Lectures complémentaires

Le Chapitre 24 du CLRS [CLRS09] traite des algorithmes de Dijkstra et de Bellman-Ford. Son Chapitre 25 couvre tous les algorithmes de plus courts chemins pour tout couple de sommets, notamment celui de Floyd-Warshall, un algorithme fondé sur une multiplication de matrices qui s'exécute en un temps $\Theta(n^3 \lg n)$, et un algorithme astucieux de Donald Johnson conçu

pour rechercher les plus courts chemins pour tout couple de sommets dans des graphes peu denses en un temps $O(n^2 \lg n + nm)$.

Lorsque les poids des arcs sont de petits entiers positifs inférieurs à une valeur $C$ connue, une implémentation plus complexe de la file de priorité dans l'algorithme de Dijkstra permet d'obtenir des temps d'exécution asymptotiques meilleurs qu'avec un tas de Fibonacci. Par exemple, Ahuja, Mehlhorn, Orlin et Tarjan [AMOT90] fondent l'algorithme de Dijkstra sur un « tas redistributif » qui permet d'obtenir un temps d'exécution $O(m + n\sqrt{\lg n})$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# 7

# ALGORITHMES SUR LES CHAÎNES DE CARACTÈRES

Une *chaîne de caractères* est une suite de caractères qui proviennent d'un jeu de caractères. Par exemple, cet ouvrage comprend des caractères qui sont des lettres, des chiffres, des signes de ponctuation et des symboles mathématiques. Ce jeu de caractères est plutôt vaste, mais il est fini. Les biologistes encodent les brins d'ADN sous forme de chaînes dont les caractères sont choisis parmi un ensemble de quatre : `A`, `C`, `G`, `T` ; ils représentent les molécules de base (adénine, cytosine, guanine et thymine).

Nous pouvons nous poser toutes sortes de questions sur les chaînes de caractères, mais, dans ce chapitre, nous nous intéresserons aux algorithmes pour trois problèmes dont les entrées sont des chaînes de caractères :

1. Trouver la plus longue sous-séquence commune à deux chaînes de caractères.
2. À partir d'un jeu d'opérations qui transforment une chaîne de caractères en une autre et du coût de chaque opération, trouver la façon la moins coûteuse de convertir une chaîne de caractères en une autre.
3. Rechercher toutes les occurrences d'un motif de caractères dans un texte.

Les deux premiers problèmes trouvent des applications en bio-informatique. Plus longue est une sous-séquence commune entre deux brins d'ADN, plus ils sont proches. Pour aligner des brins d'ADN, une solution consiste à transformer l'un en un autre ; plus le coût de la conversion est faible, plus les brins sont comparables. Le dernier problème, trouver les occurrences d'un motif à l'intérieur d'un texte, est également connu sous l'expression **recherche de sous-chaîne**. On le rencontre dans toutes sortes de programmes, par exemple chaque fois que nous utilisons une fonction de recherche sur un site web. Il est également présent en bio-informatique, car nous pouvons rechercher un brin d'ADN à l'intérieur d'un autre.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

## Plus longue sous-séquence commune

Commençons par expliquer ce que nous entendons par « séquence » et « sous-séquence ». Une **séquence** correspond à une liste d'éléments dont l'ordre est important. Un élément donné peut apparaître plusieurs fois dans une séquence. Les séquences que nous manipulerons dans ce chapitre sont des chaînes de caractères et nous utiliserons le terme « chaîne » ou « chaîne de caractères » à la place de « séquence ». De même, nous supposerons que les éléments d'une séquence sont des caractères. Par exemple, la chaîne de caractères `GACA` contient plusieurs fois le même caractère (`A`) et elle diffère de la chaîne `CAAG`, qui est constituée des mêmes caractères, mais dans un ordre différent. Une **sous-séquence Z** d'une chaîne de caractères *X* est *X* avec éventuellement des éléments retirés. Par exemple, si *X* est la chaîne `GAC`, elle possède alors huit sous-séquences : `GAC` (aucun caractère n'a été retiré), `GA` (`C` a été retiré), `GC` (`A` a été retiré), `AC` (`G` a été retiré), `G` (`A` et `C` ont été retirés), `A` (`G` et `C` ont été retirés), C (`G` et `A` ont été retirés), ainsi que la chaîne vide (tous les caractères ont été retirés). Si *X* et *Y* sont des chaînes de caractères, *Z* sera une **sous-séquence commune** à *X* et à *Y* si elle est une sous-séquence des deux. Par exemple, si *X* est la chaîne `CATCGA` et si *Y* est la chaîne `GTACCGTCA`, alors `CCA` est une sous-séquence commune à *X* et à *Y* constituée de trois caractères. Toutefois, elle n'est pas une *plus longue sous-séquence commune* (PLSC), car la plus longue sous-séquence commune `CTCA` comprend quatre caractères. `CTCA` est bien une **plus longue sous-séquence commune**, mais elle n'est pas unique, car `TCGA` en est une autre de quatre caractères. Il ne faut pas confondre les notions de sous-séquence et de sous-chaîne : une **sous-chaîne** est une sous-séquence d'une chaîne dont les caractères se trouvent à des emplacements contigus dans la chaîne. Dans le cas de la chaîne `CATCGA`, la sous-séquence `ATCG` et une sous-chaîne, contrairement à la sous-séquence `CTCA`.

Voici notre objectif : étant donné deux chaînes *X* et *Y*, nous voulons trouver la plus longue sous-séquence *Z* commune à *X* et *Y*. Pour résoudre ce problème, nous utiliserons la technique de la programmation dynamique (voir Chapitre 6).

Nous pouvons rechercher une plus longue sous-séquence commune sans recourir à la programmation dynamique, mais cette approche est déconseillée. Nous pouvons prendre chaque sous-séquence de *X* et vérifier si elle est une sous-séquence de *Y*, en partant des plus longues sous-séquences de *X* pour aller vers les plus petites, en comparant chacune à *Y* et en nous arrêtant dès que nous avons trouvé une sous-séquence commune. (Nous finirons par en trouver une car la chaîne vide est une sous-séquence de toutes les chaînes de caractères.) Si la longueur de *X* est $m$, elle comprend alors $2^m$ sous-séquences et même si nous ignorons le temps de comparaison de chaque sous-séquence avec *Y*, le temps de recherche d'une PLSC

Copyright © 2013 Dunod.

est, dans le cas le plus défavorable, au moins une fonction exponentielle de la longueur de $X$.

Au Chapitre 6, nous avons expliqué que la programmation dynamique s'appliquait lorsque nous avions une sous-structure optimale : une solution optimale à un problème comprend des solutions optimales à ses sous-problèmes. Pour rechercher une PLSC de deux chaînes de caractères à l'aide de la programmation dynamique, nous devons commencer par définir ce qui constitue un sous-problème. Les **préfixes** font de bons candidats. Si $X$ est la chaîne $x_1x_2x_3\cdots x_m$, alors le $i$-ème *préfixe* de $X$ est la chaîne de caractères $x_1x_2x_3\cdots x_i$ et nous le notons $X_i$. Dans ce cas, nous exigeons que $i$ soit dans la plage 0 à $m$ et que $X_0$ soit la chaîne vide. Par exemple, si $X$ est `CATCGA`, alors $X_4$ est `CATC`.

Nous pouvons voir qu'une PLSC de deux chaînes de caractères contient une PLSC des préfixes des deux chaînes. Prenons les chaînes $X = x_1x_2x_3\cdots x_m$ et $Y = y_1y_2y_3\cdots y_n$. Elles ont une certaine PLSC, notée $Z$, où $Z = z_1z_2z_3\cdots z_k$ pour une longueur $k$, qui peut aller de 0 au plus petit de $m$ et $n$. Que pouvons-nous en déduire à propos de $Z$ ? Examinons les derniers caractères de $X$ et de $Y$ : $x_m$ et $y_n$. Soit ils sont identiques, soit ils ne le sont pas.

- S'ils sont identiques, alors le dernier caractère de $Z$, $z_k$, doit être identique à ce caractère. Que savons-nous des autres caractères de $Z$, c'est-à-dire de $Z_{k-1} = z_1z_2z_3\cdots z_{k-1}$ ? Nous savons que $Z_{k-1}$ doit être une PLSC du reste de $X$ et de $Y$, c'est-à-dire $X_{m-1} = x_1x_2x_3\cdots x_{m-1}$ et $Y_{n-1} = y_1y_2y_3\cdots y_{n-1}$. En reprenant notre exemple précédent, où $X =$ `CATCGA`, $Y =$ `GTACCGTCA` et une PLSC est $Z =$ `CTCA`, le dernier caractère, `A`, de $X$ et de $Y$ est le dernier caractère de $Z$, et nous voyons que $Z_3 =$ `CTC` doit être une PLSC de $X_5 =$ `CATCG` et de $Y_8 =$ `GTACCGTC`.
- S'ils ne sont pas identiques, alors $z_k$ peut être identique au dernier caractère $x_m$ de $X$ ou au dernier caractère $y_n$ de $Y$, non aux deux. Ou bien, il peut être différent du dernier caractère de $X$ et de $Y$. Si $z_k$ est différent de $x_m$, nous ignorons le dernier caractère de $X$ : $Z$ doit être une PLSC de $X_{m-1}$ et de $Y$. De la même manière, si $z_k$ est différent de $y_n$, nous ignorons le dernier caractère de $Y$ : $Z$ doit être une PLSC de $X$ et de $Y_{n-1}$. En reprenant l'exemple précédent, soit $X =$ `CATCG`, $Y =$ `GTACCGTC` et $Z =$ `CTC`. Dans ce cas, $z_3$ est identique à $y_8$ (`C`) mais différent de $x_5$ (`G`). $Z$ est donc une PLSC de $X_4 =$`CATC` et de $Y$.

Ce problème possède donc une sous-structure optimale : une PLSC de deux chaînes de caractères comprend une PLSC des préfixes de deux chaînes de caractères.

Comment procédons-nous ? Nous devons résoudre un ou deux sous-problèmes, selon que les derniers caractères de $X$ et de $Y$ sont identiques ou non. S'ils le sont, nous résolvons alors un seul sous-problème – rechercher une PLSC de $X_{m-1}$ et de $Y_{n-1}$ –, puis nous ajoutons ce dernier caractère

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

pour obtenir une PLSC de $X$ et de Y. Si les derniers caractères de $X$ et $Y$ sont différents, alors nous devons résoudre deux sous-problèmes – rechercher une PLSC de $X_{m-1}$ et de $Y$, et rechercher une PLSC de $X$ de $Y_{n-1}$ –, et utiliser la plus longue des deux plus longues sous-séquences communes en tant que PLSC de $X$ et de $Y$. Si ces deux plus longues sous-séquences communes ont la même longueur, alors nous choisissons l'une ou l'autre.

Nous traitons le problème de recherche d'une PLSC de $X$ et de $Y$ en deux étapes. Premièrement, nous recherchons la longueur d'une PLSC de $X$ et de $Y$, ainsi que les longueurs des plus longues sous-séquences communes de tous les préfixes de $X$ et de $Y$. Vous pourriez être étonné que nous recherchions la longueur d'une PLSC sans connaître celle-ci. Après que nous avons obtenu les longueurs des PLSC, nous procédons à une « ingénierie inverse » du calcul des longueurs afin de rechercher une PLSC réelle de $X$ et de $Y$.

Pour que tout cela soit un peu plus précis, notons $l[i, j]$ la longueur d'une PLSC des préfixes $X_i$ et $Y_j$. La longueur d'une PLSC de $X$ et de $Y$ est donnée par $l[m,n]$. Nous pouvons démarrer les indices $i$ et $j$ à 0, car si l'un ou l'autre des préfixes a une longueur égale à 0, alors nous connaissons leur PLSC : il s'agit de la chaîne vide. Autrement dit, $l[0, j]$ et $l[i,0]$ sont égaux à 0 pour toutes les valeurs de $i$ et de $j$. Lorsque $i$ et $j$ sont positifs, nous déterminons $l[i, j]$ en examinant les plus petites valeurs de $i$ et/ou de $j$ :

- Si $i$ et $j$ sont strictement positifs et si $x_i$ est identique à $y_j$, alors $l[i, j]$ est égal à $l[i - 1, j - 1] + 1$.
- Si $i$ et $j$ sont strictement positifs et si $x_i$ est différent de $y_j$, alors $l[i, j]$ est égal au plus grand entre $l[i, j - 1]$ et $l[i - 1, j]$.

Imaginons que les valeurs de $l[i, j]$ soient stockées dans une table. Nous devons calculer ces valeurs par ordre croissant des indices $i$ et $j$. Voici la table $l[i, j]$ pour les chaînes prises en exemple (nous expliquerons plus loin les parties en grisé) :

| | | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | | G | T | A | C | C | G | T | C | A |
| $i$ | $x_i$ | $l[i, j]$ | | | | | | | | | | |
| 0 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | C | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | A | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 3 | T | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| 4 | C | | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 |
| 5 | G | | 0 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 3 |
| 6 | A | | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 |

Copyright © 2013 Dunod.

Par exemple, $l[5,8]$ vaut 3, ce qui signifie qu'une PLSC de $X_5$ =CATCG et de $Y_8$ =GTACCGTC est de longueur 3, comme nous l'avons vu à la page 123.

Pour calculer les valeurs de la table par ordre croissant des indices, nous devons, avant de calculer une entrée $l[i,j]$ particulière pour $i$ et $j$ strictement positifs, calculer les entrées $l[i,j-1]$ (immédiatement à gauche de $l[i,j]$), $l[i-1,j]$ (immédiatement au-dessus de $l[i,j]$) et $l[i-1,j-1]$ (au-dessus et à gauche de $l[i,j]$)[1]. Cette façon de calculer des entrées de la table est simple : nous pouvons les calculer ligne par ligne, de la gauche vers la droite dans chaque ligne, ou colonne par colonne, de haut en bas dans chaque colonne.

La procédure suivante traite la table comme un tableau à deux dimensions $l[0..m,0..n]$. Après qu'elle a rempli la colonne de gauche et la première ligne avec des 0, elle remplit le reste en avançant ligne par ligne :

*Procédure* CALCULER-TABLE-PLSC($X$,$Y$)
*Entrées :* $X$ et $Y$ : deux chaînes respectivement de longueur $m$ et $n$.
*Sortie :* Le tableau $l[0..m,0..n]$. La valeur de $l[m,n]$ est la longueur de la plus longue sous-séquence commune de $X$ et de $Y$.

1. Créer un tableau $l[0..m,0..n]$.
2. Pour $i = 0$ à $m$ :
   A. Fixer $l[i,0]$ à 0.
3. Pour $j = 0$ à $n$ :
   A. Fixer $l[0,j]$ à 0.
4. Pour $i = 1$ à $m$ :
   B. Pour $j = 1$ à $n$ :
      i. Si $x_i$ est identique à $y_j$, alors fixer $l[i,j]$ à $l[i-1,j-1]+1$.
      ii. Sinon ($x_i$ est différent de $y_j$), fixer $l[i,j]$ au plus grand entre $l[i,j-1]$ et $l[i-1,j]$. Si $l[i,j-1]$ est égal à $l[i-1,j]$, peu importe celui qui est choisi.
5. Retourner le tableau $l$.

Puisque le remplissage de chaque entrée de la table se fait en un temps constant et puisqu'elle contient $(m+1)\cdot(n+1)$ entrées, le temps d'exécution de CALCULER-TABLE-PLSC est $\Theta(mn)$.

Voici une bonne nouvelle : après que nous avons calculé la table $l[i,j]$, l'entrée inférieure droite, $l[m,n]$, nous donne la longueur d'une PLSC de $X$ et de $Y$. Voici la mauvaise nouvelle : aucune entrée de la table ne nous

1. Mentionner $l[i-1,j-1]$ est redondant car nous devons le calculer avant de calculer $l[i,j-1]$ et $l[i-1,j]$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

donne les caractères d'une PLSC. Nous pouvons employer la table avec les chaînes $X$ et $Y$ de façon à construire une PLSC. Cette opération demande un temps supplémentaire $O(m + n)$. Nous déterminerons comment nous avons obtenu la valeur $l[i, j]$ en procédant à une ingénierie inverse de ce calcul fondée sur $l[i, j]$ et les valeurs dont elle dépend : $x_i$, $y_j$, $l[i - 1, j - 1]$, $l[i, j - 1]$ et $l[i - 1, j]$.

Je préfère écrire cette procédure de façon récursive, en assemblant une PLSC de la fin vers le début. Lorsque la procédure trouve un caractère identique dans $X$ et dans $Y$, elle ajoute celui-ci à la fin de la PLSC en cours de construction. L'appel initial est ASSEMBLER-PLSC$(X,Y,l,m,n)$.

*Procédure* ASSEMBLER-PLSC$(X,Y,l,i,j)$

*Entrées* :

- $X$ et $Y$ : deux chaînes de caractères.
- $l$ : le tableau rempli par la procédure CALCULER-TABLE-PLSC.
- $i$ et $j$ : des indices dans $X$ et $Y$, respectivement, ainsi que dans $l$.

*Sortie :* Une PLSC de $X_i$ et de $Y_j$.

1. Si $l[i, j]$ est égal à 0, alors retourner la chaîne vide.
2. Sinon (puisque $l[i, j]$ est strictement positif, $i$ et $j$ sont strictement positifs), si $x_i$ est identique à $y_j$, alors retourner la chaîne constituée en appelant tout d'abord récursivement ASSEMBLER-PLSC $(X,Y,l,i - 1, j - 1)$, puis en ajoutant $x_i$ (ou $y_j$) à la fin de la chaîne de caractères retournée par cet appel.
3. Sinon ($x_i$ est différent de $y_j$ ), si $l[i, j - 1]$ est supérieur à $l[i - 1, j]$, alors retourner la chaîne de caractères retournée en appelant récursivement ASSEMBLER-PLSC$(X,Y,l,i, j - 1)$.
4. Sinon ($x_i$ est différent de $y_j$ et $l[i, j - 1]$ est inférieur ou égal à $l[i - 1, j]$), alors retourner la chaîne de caractères retournée en appelant récursivement ASSEMBLER-PLSC$(X,Y,l,i - 1, j)$.

Dans la table de la page 124, les entrées $l[i, j]$ en gris sont celles visitées par la récursion avec l'appel initial ASSEMBLER-PLSC$(X,Y,l,6,9)$. Les caractères $x_i$ en gris sont ceux ajoutés au cours de la construction de la PLSC. Pour nous faire une idée du fonctionnement de la procédure ASSEMBLER-PLSC, commençons avec $i = 6$ et $j = 9$. Nous déterminons que $x_6$ et $y_9$ sont tous deux le caractère `A`. Par conséquent, `A` sera le dernier caractère de la PLSC de $X_6$ et de $Y_9$, et nous poursuivons la récursion à l'étape 2. L'appel récursif se fait avec $i = 5$ et $j = 8$. Cette fois-ci, nous déterminons que $x_5$ et $y_8$ sont des caractères différents et que $l[5,7]$ est égal à $l[4,8]$. Nous poursuivons donc la récursion à l'étape 4. L'appel récursif se fait à

Copyright © 2013 Dunod.

présent avec $i = 4$ et $j = 8$. Et ainsi de suite. Si nous lisons les caractères $x_i$ grisés de haut en bas, nous obtenons la chaîne `CTCA`, qui est une PLSC. Si, lors du choix entre $l[i, j-1]$ et $l[i-1, j]$, nous avions préféré partir vers la gauche (étape 3) au lieu d'aller vers le haut (étape 4), alors la PLSC générée aurait été `TCGA`.

Comment la procédure ASSEMBLER-PLSC peut-elle prendre un temps $O(m+n)$ ? Nous observons que dans chaque appel récursif, soit $i$ soit $j$ diminue, soit les deux. Après avoir effectué $m+n$ appels récursifs, nous sommes certains que l'un de ces indices, ou les deux, atteint 0 et que la récursion s'arrête à l'étape 1.

## TRANSFORMER UNE CHAÎNE EN UNE AUTRE

Voyons à présent comment transformer une chaîne $X$ en une autre chaîne $Y$. Nous partons de $X$, que nous convertissons en $Y$, caractère par caractère. Nous supposons que $X$ et $Y$ sont respectivement constituées de $m$ et de $n$ caractères. Comme précédemment, nous faisons référence au $i$-ème caractère de chaque chaîne en utilisant le nom de celle-ci en minuscule et l'indice $i$. Par conséquent, le $i$-ème caractère de $X$ est $x_i$ et le $j$-ème caractère de $Y$ est $y_j$.

Pour transformer $X$ en $Y$, nous construisons une chaîne $Z$, de sorte qu'au terme de l'opération, $Z$ et $Y$ soient identiques. Nous gérons un indice $i$ dans $X$ et un indice $j$ dans $Z$. Nous sommes autorisés à effectuer une séquence d'opérations de transformation précises, qui peuvent altérer $Z$ et ces indices. Nous débutons avec $i$ et $j$ à 1, et nous examinons chaque caractère de $X$. Cela signifie que nous nous arrêterons uniquement lorsque $i$ aura atteint $m+1$.

Voici les opérations que nous pouvons effectuer :

- **Copier** un caractère $x_i$ de $X$ dans $Z$ en fixant $z_j$ à $x_i$ et en incrémentant ensuite $i$ et $j$.
- **Remplacer** un caractère $x_i$ de $X$ par un autre caractère $a$ en fixant $z_i$ à $a$ et en incrémentant ensuite $i$ et $j$.
- **Supprimer** un caractère $x_i$ de $X$ en incrémentant $i$, mais en ne touchant pas à $j$.
- **Insérer** un caractère dans $Z$ en fixant $z_j$ à $a$ et en incrémentant ensuite $j$, mais en ne touchant pas à $i$.

D'autres opérations sont possibles, comme intervertir deux caractères adjacents ou supprimer les caractères $x_i$ à $x_m$ en une seule opération, mais, dans notre cas, nous acceptons uniquement les opérations de copie, de remplacement, de suppression et d'insertion.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Prenons comme exemple la séquence d'opérations qui transforment la chaîne ATGATCGGCAT en la chaîne CAATGTGAATC. Les caractères grisés correspondent à $x_i$ et à $z_j$ après chaque opération :

| Opération | *X* | *Z* |
|---|---|---|
| Chaînes initiales | ATGATCGGCAT | |
| Supprimer A | ATGATCGGCAT | |
| Remplacer T par C | ATGATCGGCAT | C |
| Remplacer G par A | ATGATCGGCAT | CA |
| Copier A | ATGATCGGCAT | CAA |
| Copier T | ATGATCGGCAT | CAAT |
| Remplacer C par G | ATGATCGGCAT | CAATG |
| Remplacer G par T | ATGATCGGCAT | CAATGT |
| Copier G | ATGATCGGCAT | CAATGTG |
| Remplacer C par A | ATGATCGGCAT | CAATGTGA |
| Copier A | ATGATCGGCAT | CAATGTGAA |
| Copier T | ATGATCGGCAT | CAATGTGAAT |
| Insérer C | ATGATCGGCAT | CAATGTGAATC |

D'autres séquences d'opérations donneraient également le même résultat. Par exemple, nous pouvons simplement supprimer tous les caractères de *X* un par un, puis insérer chaque caractère de *Y* dans *Z*.

Chaque opération de transformation a un coût. Il s'agit d'une constante qui dépend non pas des caractères concernés mais uniquement de l'opération. Notre objectif est de trouver une séquence d'opérations qui transforment *X* en *Y* avec un coût total minimal. Notons le coût de l'opération de copie $c_C$, celui du remplacement $c_R$, celui de la suppression $c_S$ et celui de l'insertion $c_I$. Avec la séquence d'opérations prise en exemple, le coût total est $5c_C + 5c_R + c_S + c_I$. Nous devons supposer que le coût de $c_C$ et de $c_R$ sont inférieurs à celui de cD+cI, car, sinon, au lieu de payer $c_C$ pour copier un caractère ou de payer $c_R$ pour remplacer un caractère, nous préférerions payer $c_S + c_I$ pour supprimer le caractère et insérer ensuite ce même caractère (au lieu de le copier) ou un caractère différent (au lieu de le remplacer).

Pourquoi vouloir transformer une chaîne en une autre ? La bio-informatique nous apporte un exemple d'application. Ces biologistes alignent

Copyright © 2013 Dunod.

souvent deux séquences d'ADN afin de mesurer leur niveau de similitude. Pour aligner deux séquences $X$ et $Y$, nous alignons autant de caractères identiques que possible en insérant des espaces dans les deux séquences (y compris aux extrémités) afin que les séquences résultantes, nommons-les $X'$ et $Y'$, aient la même longueur mais ne comprennent pas des espaces aux mêmes emplacements. Autrement dit, il est impossible d'avoir une espace dans $x'_i$ et dans $y'_i$. Après que l'alignement est terminé, nous donnons des points à chaque emplacement :

- $-1$ si $x'_i$ et $y'_i$ sont identiques sans être une espace.
- $+1$ si $x'_i$ est différent de $y'_i$ sans être une espace.
- $+2$ si $x'_i$ ou $y'_i$ est une espace.

Le score d'un alignement est la somme des points de chaque emplacement individuel. Plus le score est faible, plus l'alignement des deux chaînes est proche. Nous pouvons aligner les chaînes de l'exemple précédent de la manière suivante, où ▯ représente une espace :

```
X' : ATGATCG▯GCAT▯
Y' : ▯CAAT▯GTGAATC
     *++--*-*-+--*
```

Un signe - sous un emplacement indique une note de $-1$ pour cet emplacement, un signe + indique une note de $+1$ et un symbole * indique une note de $+2$. Le score total de cet alignement est $(6 \cdot -1) + (3 \cdot 1) + (4 \cdot 2)$, c'est-à-dire 5.

Il existe plusieurs manières d'insérer des espaces et d'aligner deux séquences. Pour trouver la solution qui produit la meilleure correspondance (le score le plus faible), nous utilisons une transformation de chaînes avec les coûts $c_C = -1$, $c_R = +1$ et $c_S = c_I = +2$. Plus le nombre de caractères identiques sont mis en correspondance, meilleure est l'alignement. Le coût négatif de l'opération de copie donne avantage à la correspondance de caractères identiques. Une espace dans $Y'$ correspond à un caractère supprimé. Par conséquent, dans l'exemple précédent, le premier caractère dans $Y'$ correspond à la suppression du premier caractère (A) de $X$. Une espace dans $X'$ correspond à un caractère inséré. Par conséquent, dans l'exemple précédent, la première espace dans $X'$ correspond à l'insertion du caractère T.

Voyons comment nous pouvons transformer une chaîne $X$ en une chaîne $Y$. Nous allons utiliser la programmation dynamique, avec des sous-problèmes de la forme « convertir la chaîne de préfixe $X_i$ en la chaîne de préfixe $Y_j$ », où $i$ va de 0 à $m$ et $j$ va de 0 à $n$. Nous nommons ce sous-problème « problème $X_i \rightarrow Y_j$ », et nous débutons avec le problème $X_m \rightarrow Y_n$. Notons $coût[i, j]$ le coût d'une solution optimale au problème $X_i \rightarrow Y_j$. Par exemple, prenons $X$ =DACAAGC et $Y$ =DCCGT. Nous devons donc résoudre le problème $X_6 \rightarrow Y_4$ et nous utilisons les coûts des opérations d'alignement

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

des séquences ADN : $c_C = -1$, $c_R = +1$ et $c_S = c_I = +2$. Nous allons résoudre des sous-problèmes de la forme $X_i \rightarrow Y_j$, où $i$ va de 0 à 6 et $j$ va de 0 à 4. Ainsi, le problème $X_3 \rightarrow Y_2$ correspond à la transformation de la chaîne de préfixe $X_3$ =ACA en la chaîne de préfixe $Y_2$ =CC.

Déterminer *coût*$[I, j]$ lorsque $i$ ou $j$ vaut 0 n'a rien de compliqué car $X_0$ et $Y_0$ sont des chaînes vides. Nous convertissons une chaîne vide en $Y_j$ grâce à $j$ insertions, de sorte que *coût*$[0, j]$ soit égal à $j \cdot c_I$. De même, nous convertissons $X_i$ en une chaîne vide grâce à $i$ suppressions, de sorte que *coût*$[i,0]$ soit égal à $i \cdot c_S$. Lorsque $i$ et $j$ valent 0, nous convertissons la chaîne vide en elle-même et *coût*$[0,0]$ vaut donc 0.

Lorsque $i$ et $j$ sont strictement positifs, nous devons étudier comment la sous-structure optimale s'applique dans la transformation d'une chaîne en une autre. Supposons, pour le moment, que nous sachions quelle a été la dernière opération utilisée pour convertir $X_i$ en $Y_j$ ; il s'agit d'une des quatre opérations (copie, remplacement, suppression ou insertion).

- Si la dernière opération a été une copie, alors $x_i$ et $y_j$ devait être le même caractère. Le sous-problème restant est la conversion de $X_{i-1}$ en $Y_{j-1}$, et la solution optimale au problème $X_i \rightarrow Y_j$ doit inclure une solution optimale au problème $X_{i-1} \rightarrow Y_{j-1}$. En effet, si nous avions utilisé une solution qui n'avait pas le coût minimal pour le problème $X_{i-1} \rightarrow Y_{j-1}$, alors nous aurions employé la solution au coût minimal afin d'obtenir une meilleure solution au problème $X_i \rightarrow Y_j$. Par conséquent, en supposant que la dernière opération était une copie, nous savons que *coût*$[i, j]$ est égal à *coût*$[i-1, j-1] + c_C$.

  Dans notre exemple, examinons le problème $X_5 \rightarrow Y_3$. $x_5$ et $y_3$ sont tous deux le caractère G et si la dernière opération a consisté à copier G, alors, puisque $c_C = -1$, nous devons avoir *coût*$[5,3]$ = *coût*$[4,2] - 1$. Si *coût*$[4,2]$ vaut 4, alors *coût*$[5,3]$ doit valoir 3. Si nous pouvions trouver une solution au problème $X_4 \rightarrow Y_2$ avec un coût inférieur à 4, alors nous l'aurions utilisée pour trouver une solution au problème $X_5 \rightarrow Y_3$ avec un coût inférieur à 3.

- Si la dernière opération a été un remplacement et, dans l'hypothèse raisonnable où nous ne remplaçons pas un caractère par lui-même, alors $x_i$ et $y_j$ doivent être différents. En reprenant l'argument de la sous-structure optimale développé pour la copie, nous voyons que si la dernière opération a été un remplacement, alors *coût*$[i, j]$ est égal à *coût*$[i-1, j-1] + c_R$.

  Dans notre exemple, examinons le problème $X_5 \rightarrow Y_4$. Cette fois-ci, $x_5$ et $y_4$ sont des caractères différents (G et T, respectivement), et, par conséquent, si la dernière opération a été le remplacement de G par T, alors, puisque $c_R = +1$, nous devons avoir *coût*$[5,4]$ = *coût*$[4,3] + 1$. Si *coût*$[4,3]$ vaut 3, alors *coût*$[5,4]$ doit valoir 4.

Copyright © 2013 Dunod.

- Si la dernière opération a été une suppression, alors nous n'avons aucune contrainte sur $x_i$ ou $y_j$. La suppression peut être vue comme passer outre le caractère $x_i$ et laisser le préfixe $Y_j$ seul. Le sous-problème que nous devons résoudre est donc le problème $X_{i-1} \rightarrow Y_j$. Avec une suppression comme dernière opération, nous savons que $coût[i, j] = coût[i - 1, j] + c_S$.

  Dans notre exemple, examinons le problème $X_6 \rightarrow Y_3$. Si la dernière opération a été une suppression (le caractère supprimé doit être $x_6$, c'est-à-dire `C`), alors, puisque $c_S = +2$, nous devons avoir $coût[6,3] = coût[5,3] + 2$. Si $coût[5,3]$ vaut 3, alors $coût[6,3]$ doit valoir 5.

- Enfin, si la dernière opération a été une insertion, elle laisse $X_i$ seul mais ajoute le caractère $y_j$. Le sous-problème à résoudre est alors $X_i \rightarrow Y_{j-1}$. En supposant que la dernière opération ait été une insertion, nous savons que $coût[i, j] = coût[i, j - 1] + c_I$.

  Dans notre exemple, examinons le problème $X_2 \rightarrow Y_3$. Si la dernière opération a été une insertion (le caractère inséré doit être $y_3$, c'est-à-dire `G`), alors, puisque $c_I = +2$, nous devons avoir $coût[2,3] = coût[2,2] + 2$. Si $coût[2,2]$ vaut 0, alors $coût[2,3]$ doit valoir 2.

Bien entendu, nous ne savons pas à l'avance quelle a été la dernière opération parmi les opérations existantes. Nous voulons utiliser celle qui conduit à la plus faible valeur de $coût[i, j]$. Pour une combinaison de $i$ et de $j$, trois des quatre opérations s'appliquent. Les opérations de suppression et d'insertion s'appliquent toujours lorsque $i$ et $j$ sont tous deux strictement positifs. Par ailleurs, soit la copie soit le remplacement s'applique, selon que $x_i$ et $y_j$ sont des caractères identiques ou non. Pour calculer $coût[i, j]$ à partir d'autres valeurs de coût, nous déterminons si la copie ou le remplacement s'applique et prenons la valeur minimale de $coût[i, j]$ pour les trois opérations possibles. Autrement dit, $coût[i, j]$ est la plus petite des quatre valeurs suivantes :

- $coût[i - 1, j - 1] + c_C$, mais uniquement si $x_i$ et $y_j$ sont des caractères identiques ;
- $coût[i - 1, j - 1] + c_R$, mais uniquement si $x_i$ et $y_j$ sont des caractères différents ;
- $coût[i - 1, j] + c_S$ ;
- $coût[i, j - 1] + c_I$.

Comme nous l'avons fait pour le remplissage de la table $l$ lors du calcul d'une PLSC, nous pouvons remplir la table $coût$ ligne par ligne. En effet, chaque entrée $coût[i, j]$, avec $i$ et $j$ strictement positifs, dépend du calcul préalable des entrées immédiatement à gauche, immédiatement au-dessus et à gauche et au-dessus.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Outre la table *coût*, nous remplissons la table *op*, dans laquelle $op[i,j]$ donne la dernière opération utilisée pour convertir $X_i$ en $Y_j$. Nous pouvons remplir l'entrée $op[i,j]$ pendant que nous remplissons $coût[i,j]$. La procédure CALCULER-TABLES-TRANSFORMATION effectue le remplissage des tables *coût* et *op*, ligne par ligne, en les traitant comme des tableaux à deux dimensions.

*Procédure* CALCULER-TABLES-TRANSFORMATION$(X,Y,c_C,c_R,c_S,c_I)$

*Entrées :*

- $X$ et $Y$ : deux chaînes de caractères de longueurs respectives $m$ et $n$.
- $c_C,c_R,c_S,c_I$ : les coûts respectifs des opérations de copie, de remplacement, de suppression et d'insertion.

*Sortie :* Les tableaux $coût[0..m,0..n]$ et $op[0..m,0..n]$. La valeur de $coût[i,j]$ correspond au coût minimal de transformation du préfixe $X_i$ en préfixe $Y_j$. $coût[m,n]$ est ainsi le coût minimal de transformation de $X$ en $Y$. L'opération dans $op[i,j]$ est la dernière opération effectuée lors de la transformation de $X_i$ en $Y_j$.

1. Créer les tableaux $coût[0..m,0..n]$ et $op[0..m,0..n]$.
2. Fixer $coût[0,0]$ à 0.
3. Pour $i = 1$ à $m$ :
   A. Fixer $coût[i,0]$ à $i \cdot c_S$ et fixer $op[i,0]$ à *supprimer* $x_i$.
4. Pour $j = 1$ à $n$ :
   A. Fixer $coût[0,j]$ à $j \cdot c_I$ et fixer $op[0,j]$ à *insérer* $y_j$.
5. Pour $i = 1$ à $m$ :
   A. Pour $j = 1$ à $n$ :
      (Déterminer quelle opération entre la copie et le remplacement s'applique, et fixer $coût[i,j]$ et $op[i,j]$ en fonction de l'opération, parmi les trois, qui donne la valeur la plus faible à $coût[i,j]$.)
      i. Fixer $coût[i,j]$ et $op[i,j]$ de la manière suivante :
         a. Si $x_i$ et $y_j$ sont identiques, alors fixer $coût[i,j]$ à $coût[i-1,j-1] + c_C$ et fixer $op[i,j]$ à *copier* $x_i$.
         b. Sinon ($x_i$ et $y_j$ sont différents), fixer $coût[i,j]$ à $coût[i-1,j-1] + c_R$ et fixer $op[i,j]$ à *remplacer* $x_i$ *par* $y_j$.
      ii. Si $coût[i-1,j] + c_S < coût[i,j]$, alors fixer $coût[i,j]$ à $coût[i-1,j] + c_S$ et fixer $op[i,j]$ à *supprimer* $x_i$.
      iii. Si $coût[i,j-1] + c_I < coût[i,j]$, alors fixer $coût[i,j]$ à $coût[i,j-1] + c_I$ et fixer $op[i,j]$ à *insérer* $x_j$.
6. Retourner les tableaux *coût* et *op*.

Les tables *coût* et *op* calculées par la procédure CALCULER-TABLES-TRANSFORMATION pour notre exemple de transformation de $X =$ ACAAGC en

Copyright © 2013 Dunod.

$Y$ =CCGT avec $c_C = -1$, $c_R = +1$ et $c_S = c_I = +2$ sont données ci-après. À l'intersection de la ligne $i$ et de la colonne $j$, nous trouvons les valeurs de *coût*$[i, j]$ et de *op*$[i, j]$, avec les noms des opérations en abrégé. Par exemple, la dernière opération utilisée lors de la transformation de $X_5$ = ACAAG en $Y_2$ =CC a été le remplacement de G par C, et une séquence optimale des opérations de transformation de ACAAG en CC présente un coût total de 6.

| | | $j$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|
| | | $y_j$ | | C | C | G | T |
| $i$ | $x_i$ | | | | | | |
| 0 | | | 0 | 2<br>*ins* C | 4<br>*ins* C | 6<br>*ins* G | 8<br>*ins* T |
| 1 | A | | 2<br>*sup* A | 1<br>*rep* A *par* C | 3<br>*rep* A *par* C | 5<br>*rep* A *par* G | 7<br>*rep* A *par* T |
| 2 | C | | 4<br>*sup* C | 1<br>*cop* C | 0<br>*cop* C | 2<br>*ins* G | 4<br>*ins* T |
| 3 | A | | 6<br>*sup* A | 3<br>*sup* A | 2<br>*rep* A *par* C | 1<br>*rep* A *par* G | 3<br>*rep* A *par* T |
| 4 | A | | 8<br>*sup* A | 5<br>*sup* A | 4<br>*rep* A *par* C | 3<br>*rep* A *par* G | 2<br>*rep* A *par* T |
| 5 | G | | 10<br>*sup* G | 7<br>*sup* G | 6<br>*rep* G *par* C | 3<br>*cop* G | 4<br>*rep* G *par* T |
| 6 | C | | 12<br>*sup* C | 9<br>*cop* C | 6<br>*cop* C | 5<br>*sup* C | 4<br>*rep* C *par* T |

La procédure CALCULER-TABLES-TRANSFORMATION remplit chaque entrée des tables en un temps constant, tout comme le faisait CALCULER-TABLE-PLSC. Puisque chaque table comprend $(m + 1) \cdot (n + 1)$ entrées, CALCULER-TABLES-TRANSFORMATION s'exécute en un temps $\Theta(mn)$.

Pour construire la séquence des opérations qui transforment $X$ en $Y$, nous consultons la table *op* en partant de la dernière entrée *op*$[m,n]$. Nous procédons par récursion, à la manière de la procédure ASSEMBLER-PLSC, en ajoutant chaque opération rencontrée dans la table *op* à la fin de la séquence des opérations. La procédure ASSEMBLER-TRANSFORMATION est donnée ci-après. L'appel initial est ASSEMBLER-TRANSFORMATION $(op,m,n)$.

*Procédure* ASSEMBLER-TRANSFORMATION$(op,i,j)$
*Entrées :*

- *op* : la table des opérations remplie par la procédure CALCULER-TABLES-TRANSFORMATION.
- $i$ et $j$ : des indices dans la table *op*.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

*Sortie :* Une séquence d'opérations qui transforment la chaîne $X$ en la chaîne $Y$, où $X$ et $Y$ sont les chaînes de caractères données en entrée de la procédure CALCULER-TABLES-TRANSFORMATION.

1. Si $i$ et $j$ sont égaux à 0, alors retourner une séquence vide.
2. Sinon (au moins $i$ ou $j$ est strictement positif), faire la chose suivante :
   A. Si $op[i, j]$ est une copie ou un remplacement, alors retourner la séquence constituée par le retour de l'appel récursif à ASSEMBLER-TRANSFORMATION$(op, i - 1, j - 1)$ auquel est ajouté $op[i, j]$.
   B. Sinon ($op[i, j]$ n'est pas une opération de copie ou de remplacement), alors si $op[i, j]$ est une opération de suppression, retourner la séquence constituée par le retour de l'appel récursif à ASSEMBLER-TRANSFORMATION$(op, i - 1, j)$ auquel est ajouté $op[i, j]$.
   C. Sinon ($op[i, j]$ n'est pas une opération de copie, de remplacement ou de suppression), retourner la séquence constituée par le retour de l'appel récursif à ASSEMBLER-TRANSFORMATION$(op, i, j - 1)$ auquel est ajouté $op[i, j]$.

Voici la séquence des opérations qui convertit $X =$ `ACAAGC` en une chaîne de caractères $Z$ identique à $Y =$ `CCGT`.

| Opération | *X* | *Z* |
|---|---|---|
| Chaînes initiales | ACAAGC | |
| supprimer `A` | ACAAGC | |
| copier `C` | ACAAGC | C |
| supprimer `A` | ACAAGC | C |
| remplacer `A` par `C` | ACAAGC | CC |
| copier `G` | ACAAGC | CCG |
| remplacer `C` par `T` | ACAAGC | CCGT |

À l'instar d'ASSEMBLER-PLSC, chaque appel récursif de la procédure ASSEMBLER-TRANSFORMATION diminue $i$ ou $j$, ou les deux. La récursion s'arrête donc après au plus $m + n$ appels récursifs. Puisque chaque appel se fait en un temps constant avant et après la récursion, la procédure ASSEMBLER-TRANSFORMATION s'exécute en un temps $O(m + n)$.

Une subtilité de la procédure ASSEMBLER-TRANSFORMATION mérite qu'on s'y attarde. La récursion s'arrête uniquement lorsque $i$ et $j$ sont arrivés à 0. Supposons que $i$ ou $j$, non les deux, soit égal à 0. Chacun des trois

Copyright © 2013 Dunod.

cas des étapes 2A, 2B et 2C déclenche un appel récursif avec la valeur de $i$ ou de $j$, ou les deux, diminuée de 1. Peut-on rencontrer un appel récursif dans lequel $i$ ou $j$ a la valeur $-1$ ? Heureusement, la réponse est non. Supposons que $j = 0$ et que $i$ soit strictement positif lors d'un appel à ASSEMBLER-TRANSFORMATION. En raison de la construction de la table $op$, $op[i,0]$ est une suppression et l'étape 2B est exécutée. Dans cette étape, l'appel récursif est ASSEMBLER-TRANSFORMATION$(op, i-1, j)$ et la valeur de $j$ dans l'appel reste donc 0. De manière comparable, si $i = 0$ et si $j$ est strictement positif, alors $op[0,j]$ est une insertion et l'étape 2C s'exécute. Dans l'appel récursif à ASSEMBLER-TRANSFORMATION$(op, i, j-1)$, la valeur de $i$ reste 0.

## RECHERCHE DE CHAÎNE DE CARACTÈRES

Dans le problème de recherche de chaîne de caractères, nous avons deux chaînes : une **chaîne de texte** $T$ et une **chaîne de motif** $P$. L'objectif est de trouver toutes les occurrences de $P$ dans $T$. Nous simplifierons la rédaction en utilisant « texte » et « motif », et nous supposerons que le texte et le motif sont constitués respectivement de $n$ et de $m$ caractères, avec $m \leq n$ (puisque la recherche d'un motif plus long que le texte n'a pas de sens). Nous noterons $p_1 p_2 p_3 \ldots p_m$ les caractères de $P$ et $t_1 t_2 t_3 \ldots t_n$ ceux de $T$.

Puisque nous voulons rechercher toutes les occurrences du motif $P$ dans le texte $T$, une réponse correspond au nombre de décalages de $P$ qui permettent de trouver $P$ dans $T$. En d'autres termes, nous disons que $P$ apparaît au décalage $s$ dans le texte $T$ si la sous-chaîne de $T$ qui commence à $t_{s+1}$ est identique au motif $P$ : $t_{s+1} = p_1$, $t_{s+2} = p_2$, etc., jusqu'à $t_{s+m} = p_m$. Le plus petit décalage possible est 0. Et, puisque le motif ne doit pas s'étendre au-delà de la fin du texte, le plus grand décalage possible est $n - m$. Nous voulons connaître tous les décalages qui permettent de trouver $P$ dans $T$. Par exemple, si le texte $T$ est `GTAACAGTAAACG` et si le motif $P$ est `AAC`, alors $P$ apparaît dans $T$ aux décalages 2 et 9.

Pour savoir si le motif $P$ apparaît dans le texte $T$ à un décalage $s$, nous devons comparer les $m$ caractères de $P$ aux caractères de $T$. En supposant qu'il faille un temps constant pour comparer un seul caractère de $P$ à un seul caractère de $T$, il faut un temps $\Theta(m)$ pour comparer les $m$ caractères dans le cas le plus défavorable. Bien entendu, dès qu'un caractère de $P$ ne correspond pas à un caractère de $T$, il est inutile de poursuivre la comparaison des caractères suivants. Le cas le plus défavorable se rencontre à chaque décalage pour lequel $P$ apparaît dans $T$.

Il est plutôt facile de comparer simplement le motif au texte pour chaque décalage possible, en allant de 0 à $n - m$. Voici comment cette approche fonctionnerait pour la recherche du motif `AAC` dans le texte `GTAACAGTAAACG`,

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

pour chaque décalage possible (les caractères qui correspondent sont en gris) :

| Décalage | Texte et motif |
|---|---|
| 0 | GTAACAGTAAACG<br>AAC |
| 1 | GTAACAGTAAACG<br>AAC |
| 2 | GTAACAGTAAACG<br>AAC |
| 3 | GTAACAGTAAACG<br>AAC |
| 4 | GTAACAGTAAACG<br>AAC |
| 5 | GTAACAGTAAACG<br>AAC |
| 6 | GTAACAGTAAACG<br>AAC |
| 7 | GTAACAGTAAACG<br>AAC |
| 8 | GTAACAGTAAACG<br>AAC |
| 9 | GTAACAGTAAACG<br>AAC |
| 10 | GTAACAGTAAACG<br>AAC |

Cette solution est plutôt inefficace : avec $n - m + 1$ décalages possibles, chacun prenant un temps de comparaison $O(m)$, le temps d'exécution total est $O((n - m)m)$. Nous avons à examiner pratiquement chaque caractère du texte à $m$ reprises.

Il est possible de faire mieux que cela, car cette méthode simple ignore une information importante. Dans l'exemple précédent, l'étude du décalage $s = 2$ nous conduit à examiner tous les caractères de la sous-chaîne $t_3t_4t_5$ =AAC. Pourtant, au décalage suivant, $s = 3$, nous examinons à nouveau $t_4$ et $t_5$. Il serait plus efficace de ne pas revenir, si possible, sur ces caractères. Voyons

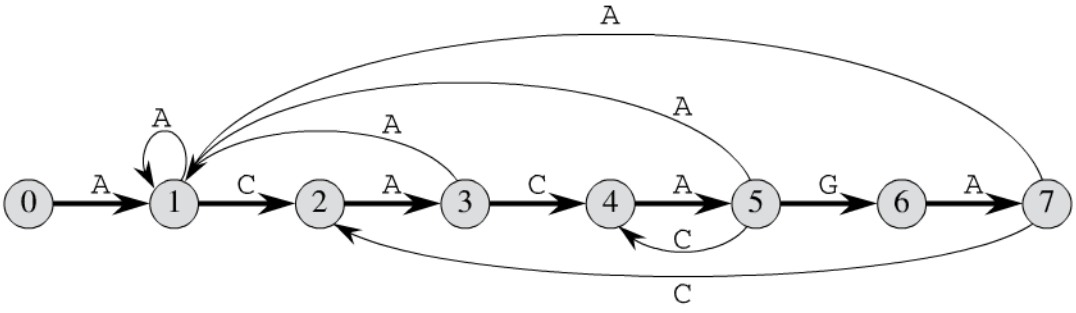
Copyright © 2013 Dunod.

une approche plus astucieuse de la recherche d'une chaîne qui évite de perdre du temps à examiner de façon répétitive le texte. Au lieu d'examiner à $m$ reprises les caractères du texte, elle examine chaque caractère exactement une fois.

L'approche la plus efficace se fonde sur un **automate fini**. Bien que le nom puisse impressionner, le concept sous-jacent est relativement simple. Les applications des automates finis sont nombreuses, mais nous nous intéresserons uniquement à la recherche de chaîne de caractères. Un automate fini est constitué d'états, avec des **transitions** qui permettent de passer d'un état à l'autre en fonction de la séquence des caractères en entrée. L'automate fini commence dans un **état** précis et consomme les caractères fournis en entrée, un à la fois. En fonction de l'état dans lequel il se trouve et du caractère qu'il lit, il passe dans un nouvel état particulier.

Dans notre application de recherche d'une chaîne de caractères, la séquence d'entrée correspond aux caractères du texte $T$ et l'automate fini comprend $m + 1$ états, c'est-à-dire 1 plus le nombre de caractères du motif $P$, numérotés de 0 à $m$. (L'automate est dit « fini » car son nombre d'états est fini.) L'automate fini commence dans l'état 0. Lorsqu'il se trouve dans l'état $k$, les $k$ derniers caractères du texte qu'il a lus correspondent aux $k$ premiers caractères du motif. Lorsque l'automate arrive dans l'état $m$, il a rencontré l'intégralité du motif dans le texte.

Prenons un exemple en utilisant uniquement les caractères `A`, `C`, `G` et `T`. Supposons que le motif soit `ACACAGA`, avec $m = 7$ caractères. Voici l'automate fini correspondant, avec les états 0 à 7 :




Les cercles représentent les états et les flèches, libellées par les caractères, montrent les transitions d'un état à l'autre en fonction du caractère d'entrée. Par exemple, les flèches qui partent de l'état 5 sont libellées `A`, `C` et `G`. La flèche vers l'état 1, libellée `A`, signifie que, lorsque l'automate fini se trouve dans l'état 5 et qu'il lit le caractère de texte `A`, il passe à l'état 1. De la même manière, la flèche vers l'état 4, libellée `C`, indique que, lorsque l'automate fini se trouve dans l'état 5 et qu'il lit le caractère `C`, il passe à l'état 4. Vous remarquerez que les flèches de la « dorsale » horizontale de l'automate fini sont plus épaisses et que les libellés associés, lus de gauche à droite, donnent le motif `ACACAGA`. Lorsque le motif est présent dans le texte, l'automate fini se déplace d'un état vers la droite pour chaque caractère,

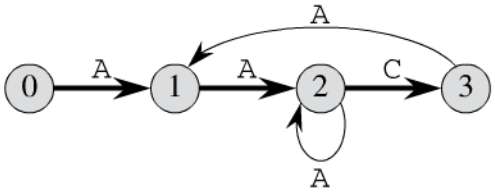
Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

jusqu'à atteindre le dernier état, où il déclare avoir trouvé une occurrence du motif dans le texte. Notez également que certaines flèches sont absentes, par exemple aucune n'est libellée T. Dans ce cas, la transition correspondante mène à l'état 0.

De façon interne, l'automate gère une table *état-suivant*, qui est indicée par tous les états et tous les caractères d'entrée possibles. La valeur de $état - suivant[s,a]$ donne le numéro de l'état dans lequel aller lorsque l'automate fini se trouve dans l'état $s$ et qu'il lit le caractère $a$ sur l'entrée. Voici l'intégralité de la table *état-suivant* pour le motif ACACAGA :

| État | Caractère | | | |
|---|---|---|---|---|
| | A | C | G | T |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 |
| 2 | 3 | 0 | 0 | 0 |
| 3 | 1 | 4 | 0 | 0 |
| 4 | 5 | 0 | 0 | 0 |
| 5 | 1 | 4 | 6 | 0 |
| 6 | 7 | 0 | 0 | 0 |
| 7 | 1 | 2 | 0 | 0 |

L'automate fini se déplace d'un état vers la droite pour chaque caractère qui correspond au motif, et vers la gauche, ou reste à la même place ( $état - suivant[1,A]$ vaut 1), pour chaque caractère qui ne correspond pas au motif. Nous verrons plus loin comment construire la table *état-suivant*. Pour le moment, voyons comment se comporte l'automate fini pour le motif AAC et le texte d'entrée GTAACAGTAAACG. Voici cet automate :




À partir de ce tracé, nous pouvons déduire la table *état-suivant* :

| État | Caractère | | | |
|---|---|---|---|---|
| | A | C | G | T |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 2 | 0 | 0 | 0 |
| 2 | 2 | 3 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 |

Copyright © 2013 Dunod.

Voici les états dans lesquels passe l'automate fini et les caractères de texte lus pour déclencher ces transitions :

| **État** | **0** | **0** | **0** | **1** | **2** | **3** | **1** | **0** | **0** | **1** | **2** | **2** | **3** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Caractère** | G | T | A | A | C | A | G | T | A | A | A | C | G | |

J'ai grisé les deux fois où l'automate fini arrive à l'état 3, car cela signifie qu'il a trouvé une occurrence du motif AAC.

Voici la procédure RECHERCHER-CHAÎNE-AUTOMATE-FINI. Elle suppose que la table soit déjà construite.

*Procédure* RECHERCHER-CHAÎNE-AUTOMATE-FINI ($T$,*état* − *suivant*,$m$,$n$)

*Entrées* :

- $T$, $n$ : une chaîne de texte et sa longueur.
- *état-suivant* : la table des transitions entre états, construite à partir du motif recherché.
- $m$ : la longueur du motif. Les lignes de la table *état-suivant* sont indicées de 0 à $m$, et les colonnes par les caractères que peut contenir le texte.

*Sortie* : Afficher les décalages auxquels le motif est trouvé dans le texte.

1. Fixer *état* à 0.
2. Pour $i = 1$ à $n$ :
   A. Fixer *état* à la valeur de *état* − *suivant*[*état*,*ti*].
   B. Si *état* est égal à $m$, afficher « Motif rencontré au décalage $i - m$ ».

Si nous exécutons RECHERCHER-CHAÎNE-AUTOMATE-FINI avec l'exemple précédent, dans lequel $m$ est égal à 3, l'automate fini atteint l'état 3 après avoir lu les caractères $t_5$ et $t_{12}$. Par conséquent, la procédure va afficher « Motif rencontré au décalage 2 » (2 = 5 – 3) et « Motif rencontré au décalage 9 » (9 = 12 – 3).

Puisque chaque itération de la boucle de l'étape 2 prend un temps constant et que la boucle effectue $n$ itérations, nous déterminons sans difficulté que le temps d'exécution de RECHERCHER-CHAÎNE-AUTOMATE-FINI est $\Theta(n)$.

Voilà pour la partie facile. La partie difficile correspond à la construction de la table *état-suivant* de l'automate fini à partir d'un motif donné. Rappelons l'idée sous-jacente :

> Lorsque l'automate fini se trouve dans l'état $k$, alors les $k$ derniers caractères lus sont les $k$ premiers caractères du motif.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Pour rendre cette idée plus concrète, revenons à l'automate fini de la page 137 ; il correspond au motif ACACAGA. Voyons pourquoi *état − suivant*[5,*C*] vaut 4. Si l'automate fini est arrivé dans l'état 5, alors les cinq derniers caractères qu'il a lus à partir du texte sont ACACA, ce que nous pouvons voir en examinant la dorsale de l'automate fini. Si le caractère suivant est C, il ne correspond pas au motif et l'automate fini ne peut pas poursuivre vers l'état 6. Cependant, il n'a pas à retourner vers l'état 0. En effet, les quatre derniers caractères lus sont à présent ACAC, qui correspondent aux quatre premiers caractères du motif ACACAGA. C'est pourquoi, lorsque l'automate fini se trouve dans l'état 5 et qu'il lit le caractère C, il passe à l'étape 4 ; les derniers caractères rencontrés sont les quatre premiers du motif.

Nous venons de décrire en partie la règle de construction de la table *état-suivant*, mais nous devons donner deux définitions avant de nous y atteler. Tout d'abord, rappelons que pour $i$ dans la plage 0 à $m$, le préfixe $P_i$ du motif $P$ est la sous-chaîne constituée des $i$ premiers caractères de $P$. (Lorsque $i$ est égal à 0, le préfixe est la chaîne vide.) De façon comparable, définissons un **suffixe** du motif comme une sous-chaîne de caractères à partir de la fin de $P$. Par exemple, AGA est un suffixe du motif ACACAGA. Définissons également la **concaténation** d'une chaîne $X$ et d'un caractère $a$ comme la nouvelle chaîne de caractères formée par l'ajout de $a$ à la fin de $X$ et notons-la $Xa$. Par exemple, la concaténation de la chaîne CA et du caractère T donne la chaîne CAT.

Nous sommes enfin prêts à construire *état* − *suivant*[$k$,$a$], où $k$ est le numéro d'un état de 0 à $m$ et $a$ est un caractère qui peut apparaître dans le texte. Lorsque nous sommes à l'état $k$, nous avons rencontré le préfixe $P_k$ dans le texte. Autrement dit, les $k$ derniers caractères du texte lus sont identiques aux $k$ premiers caractères du motif. Lorsque nous lisons le caractère suivant, nommé $a$, nous avons vu $P_k a$ (la concaténation de $P_k$ et de $a$) dans le texte. À ce stade, quelle est la longueur d'un préfixe de $P$ que nous venons de rencontrer ? Voici une autre manière de poser cette question : quelle est la longueur d'un préfixe de $P$ qui apparaît à la fin de $P_k a$ ? Cette longueur sera le numéro de l'état suivant.

Récapitulons :

> Prendre le préfixe $P_k$ (les $k$ premiers caractères de $P$) et le concaténer avec le caractère $a$. Nommer la chaîne résultante $P_k a$. Trouver le plus long préfixe de $P$ qui soit aussi un suffixe de $P_k a$. *état* − *suivant*[$k$,$a$] est alors la longueur de ce plus long préfixe.

Nous manipulons effectivement quelques préfixes et quelques suffixes. Voyons donc comment nous déterminons que *état* − *suivant*[5,*C*] vaut 4 avec le motif $P$ = ACACAGA. Puisque $k$ vaut 5 dans cet exemple, nous prenons le préfixe $P_5$, c'est-à-dire ACACA, et le concaténons avec le caractère C, ce qui donne ACACAC. Nous voulons trouver le plus long préfixe de ACACAGA

Copyright © 2013 Dunod.

qui soit également un suffixe de ACACACP. Puisque la longueur de la chaîne ACACAC est 6 et qu'un suffixe ne peut pas être plus long que la chaîne dont il est un suffixe, nous pouvons commencer notre examen à $P_6$ et aller vers des préfixes de plus en plus courts. Dans notre cas, $P_6$ est ACACAG, qui n'est pas un suffixe de ACACAC. Nous examinons donc $P_5$, c'est-à-dire ACACA, qui n'est pas plus un suffixe de ACACAC. Nous passons donc à $P_4$, qui vaut ACAC. Puisqu'il s'agit à présent d'un suffixe de ACACAC, nous arrêtons la procédure et déterminons que $état - suivant[5,C]$ doit être égal à 4.

Vous vous demandez peut-être si nous pouvons toujours trouver un préfixe de $P$ qui soit également un suffixe de $P_k a$. La réponse est oui, car la chaîne vide est un préfixe et un suffixe de n'importe quelle chaîne. Lorsque le plus long préfixe de $P$ qui est également un suffixe de $P_k a$ se révèle être la chaîne vide, nous fixons $état - suivant[k,a]$ à 0. En reprenant le motif $P$ = ACACAGA, voyons comment déterminer $état - suivant[3,G]$. La concaténation de $P_3$ et de G donne la chaîne ACAG. Nous examinons les préfixes de $P$, en commençant par $P_4$ (puisque la longueur de ACAG est 4) et en allant vers les préfixes plus courts. Aucun des préfixes ACAC, ACA, AC et A n'est un suffixe de ACAG. Par conséquent, nous établissons que la chaîne vide est le plus long préfixe possible. Puisque la chaîne vide a une longueur égale à 0, nous fixons $état - suivant[3,G]$ à 0.

Combien de temps faut-il pour remplir l'intégralité de la table *état-suivant* ? Nous savons qu'elle comprend une ligne pour chaque état de l'automate fini. Elle a donc $m + 1$ lignes, numérotées de 0 à $m$. Le nombre de colonnes dépend du nombre de caractères que peut contenir le texte ; appelons ce nombre $q$. La table *état-suivant* comprend donc $q(m + 1)$ entrées. Pour remplir une entrée $état - suivant[k,a]$, nous effectuons les opérations suivantes :

1. Construire la chaîne $P_k a$.
2. Fixer $i$ à la plus petite valeur entre $k + 1$ (la longueur de $P_k a$) et $m$ (la longueur de $P$).
3. Tant que $P_i$ n'est pas un suffixe de $P_k a$, faire la chose suivante :
   A. Fixer $i$ à $i - 1$.

Nous ne connaissons pas à l'avance le nombre d'exécutions de la boucle de l'étape 3, mais nous savons qu'elle effectue au plus $m + 1$ itérations. De même, nous ne connaissons pas à l'avance le nombre de caractères de $P_i$ et de $P_k a$ que nous devons tester à l'étape 3, mais nous savons qu'il en existe toujours au plus $i$, qui vaut au plus $m$. Puisque la boucle effectue au maximum $m + 1$ itérations et que chacune vérifie au maximum $m$ caractères, le remplissage de $état - suivant[k,a]$ prend un temps $O(m^2)$. Puisque la table *état-suivant* comprend $q(m + 1)$ entrées, le temps total pour la remplir est donc $O(m^3 q)$.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Dans la pratique, le temps de remplissage de la table *état-suivant* n'est pas si mauvais. J'ai programmé l'algorithme de recherche de chaînes en C++ sur mon Mac-Book Pro à 2,4 GHz et l'ai compilé avec le niveau d'optimisation -O3. Je lui ai donné le motif `a man, a plan, a canal, panama`, avec pour alphabet les 128 caractères du jeu ASCII. Le programme a construit une table *état-suivant* de 31 lignes et 127 colonnes (j'ai retiré la colonne du caractère `null`) en approximativement 1,35 millisecondes. Avec un motif plus court, le programme est évidemment plus rapide : il faut environ 0,07 milliseconde pour créer la table qui correspond au motif `panama`.

Quoi qu'il en soit, certaines applications utilisent très fréquemment la recherche de chaînes et, dans ce cas, le temps $O(m^3q)$ de construction de la table *état-suivant* pourrait poser un problème. Je n'entrerai pas dans les détails, mais il existe une solution pour arriver à un temps $\Theta(mq)$. En réalité, il est possible de faire mieux. L'algorithme KMP (conçu par Knuth, Morris et Pratt) se fonde sur un automate fini mais évite la création et le remplissage de la table *état-suivant*. À la place, il utilise un tableau *aller-vers* de $m$ numéros d'états qui permet à l'automate fini de simuler l'existence d'une table *état-suivant*. Le remplissage du tableau *aller-vers* se fait en un temps $\Theta(m)$. Cet algorithme est trop complexe pour le présenter, mais je l'ai programmé sur mon MacBook Pro. Appliqué au motif `a man, a plan, a canal, panama`, il lui a fallu autour d'une microseconde pour construire le tableau *aller-vers*. Pour le motif plus court `panama`, 600 nanosecondes (0,000 000 6 seconde) ont suffi. Plutôt pas mal ! À l'instar de la procédure RECHERCHER-CHAÎNE-AUTOMATE-FINI, l'algorithme KMP prend un temps $\Theta(n)$ pour effectuer la correspondance entre le motif et le texte, après qu'il a construit le tableau *aller-vers*.

Copyright © 2013 Dunod.

## LECTURES COMPLÉMENTAIRES

Le Chapitre 15 du CLRS [CLRS09] détaille la programmation dynamique, notamment la recherche d'une plus longue sous-séquence commune. L'algorithme de transformation d'une chaîne en une autre présenté dans ce chapitre donne une partie de la solution d'un problème posé au Chapitre 15 du CLRS. (Ce problème comprend les deux opérations non prises en compte dans ce chapitre : échanger des caractères adjacents et supprimer un suffixe de $X$. Mais vous ne pensiez tout de même pas que j'allais mécontenter mes coauteurs en donnant la solution.)

Les algorithmes de recherche de chaînes de caractères font l'objet du Chapitre 32 du CLRS. Il décrit l'algorithme fondé sur un automate fini, ainsi que l'algorithme KMP. La première édition de l'ouvrage *Algorithmique* [CLR90] présentait l'algorithme de Boyer-Moore, qui se révèle particulièrement efficace lorsque le motif est long et le nombre de caractères de l'alphabet, élevé.

# 8 Bases de la cryptographie

Lorsque nous achetons un produit sur Internet, nous devons en général donner le numéro de notre carte bancaire au serveur du site web du marchand ou à celui d'un service de paiement tiers. Pour que le numéro de la carte arrive au serveur, il est transmis sur Internet. Internet est un réseau public et tout le monde peut examiner les données qui y transitent. Par conséquent, si notre numéro de carte bancaire transite par Internet sans être camouflé d'une manière ou d'une autre, n'importe qui peut le trouver et acheter des biens et des services à l'aide de notre compte.

Cela dit, il est peu probable que quelqu'un s'asseye en attendant que nous envoyions sur Internet des données qui ressemblent à un numéro de carte bancaire. Il est plus probable que cette personne attende que n'importe qui le fasse et, si cela tombe sur nous, nous serons la victime malchanceuse. La meilleure solution serait que nous masquions notre numéro de carte bancaire dès qu'il doit passer par Internet. D'ailleurs, c'est probablement le cas. Si nous utilisons un site web sécurisé, ceux dont l'URL commence par `https :` à la place de `http :`, alors notre navigateur camoufle les informations envoyées à l'aide d'un processus de **chiffrement**. (Le protocole `https` met également en place une authentification afin que nous sachions que nous nous connectons au site auquel nous voulions nous connecter.) Dans ce chapitre, nous allons présenter le chiffrement ainsi que la procédure inverse, le **déchiffrement**, dans lequel des informations chiffrées sont reconverties sous leur forme d'origine. Les procédures de chiffrement et de chiffrement constituent les bases de la cryptographie.

Même si je considère que mon numéro de carte bancaire est une information importante à protéger, je reconnais qu'en fin de compte il n'est pas aussi important que cela. Si ce numéro m'est volé, la sécurité nationale n'est pas en jeu. En revanche, si une personne est capable de jeter un œil sur les instructions envoyées par le ministre des affaires étrangères à un diplomate ou si elle peut intercepter des informations militaires, les risques

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

sont différents. Par conséquent, non seulement nous devons trouver des mécanismes pour chiffrer et déchiffrer des informations, mais ils doivent également être difficiles à mettre en échec.

Dans ce chapitre, nous décrivons les idées sous-jacentes au chiffrement et au déchiffrement. La cryptographie moderne va plus loin, bien plus loin que je ne peux l'expliquer ici. N'essayez pas de développer un système sécurisé uniquement à partir du contenu de ce chapitre ; si cela vous intéresse, vous devrez comprendre la cryptographie moderne de manière beaucoup plus approfondie pour que votre système soit sûr, en théorie comme en pratique. Par exemple, vous devez suivre les standards établis, tels que ceux publiés par le NIST (*National Institute of Standards and Technology*). Comme me l'a indiqué Ron Rivest (l'un des inventeurs du cryptosystème RSA décrit plus loin dans ce chapitre), « En général, la cryptographie est comparable à un tournoi d'arts martiaux et, pour l'employer en pratique, vous devez comprendre les dernières techniques de l'adversaire ». Cependant, ce chapitre vous donnera un avant-goût des algorithmes qui ont été motivés par le besoin de chiffrer et de déchiffrer des informations.

En cryptographie, l'information d'origine est appelée **texte en clair** et sa version chiffrée, **texte chiffré**. Par conséquent, le chiffrement a pour objectif de transformer le texte en clair en un texte chiffré, tandis que le déchiffrement convertit un texte chiffré vers le texte en clair d'origine. L'information nécessaire aux conversions est la *clé* cryptographique.

## Chiffrement par simple substitution

Dans le cas d'un **chiffrement par simple substitution**, un texte est chiffré en remplaçant une lettre par une autre. Le déchiffrement du texte chiffré se fait en procédant à la substitution inverse. Jules César communiquait avec ses généraux en utilisant un **chiffrement par décalage**, dans lequel l'expéditeur remplaçait chaque lettre du message par celle qui se trouvait à trois positions plus loin dans l'alphabet, avec un retour au début pour les dernières lettres. Par exemple, avec notre alphabet de vingt-six lettres, *A* serait remplacé par *D* et *Y* par *B* (après *Y*, nous avons *Z*, puis *A* et *B*). Dans le chiffre de César, si un général avait besoin de troupes plus nombreuses, il pouvait envoyer le texte en clair « Il me faut une centaine de soldats » sous la forme du texte chiffré « Lo ph idxw xqh fhqwdlqh gh vrogdwv ». À réception de ce texte chiffré, César remplaçait chaque lettre par la lettre qui se trouve à trois positions avant dans l'alphabet, avec un retour à la fin pour les premières. (En ce temps-là, le message aurait évidemment été rédigé en latin, avec l'alphabet correspondant.)

Copyright © 2013 Dunod.

Si nous interceptons un message que nous savons chiffré par un système de décalage, il est extrêmement facile à déchiffrer, même sans connaître à l'avance ce décalage (la clé). Il suffit d'essayer toutes les possibilités jusqu'à ce que la forme en clair du texte chiffré ait un sens. Pour un alphabet de vingt-six caractères, il suffit d'essayer vingt-cinq décalages.

Le chiffrement peut être un tantinet plus sûr en convertissant chaque caractère en un autre caractère unique, mais pas nécessairement celui qui apparaît à un nombre de positions fixe plus loin dans l'alphabet. Autrement dit, nous pouvons établir une transposition des caractères et l'utiliser comme clé. Cela reste un chiffrement par simple substitution, mais il est meilleur qu'un chiffrement par décalage. Si nous avons un jeu de $n$ caractères, l'espion qui intercepte un message doit retrouver la transposition que nous avons utilisée parmi les $n!$ (factorielle $n$) possibles. La fonction factorielle augmente très rapidement avec $n$, plus rapidement même qu'une fonction exponentielle.

Dans ce cas, pourquoi ne pas simplement convertir chaque caractère en un autre ? Si vous avez déjà essayé de résoudre les cryptogrammes publiés dans certains magazines, vous savez que vous pouvez utiliser les fréquences et les combinaisons des lettres pour réduire les possibilité. Supposons que le texte en clair « Il me faut une centaine de soldats » devienne « Sc jb lkfd ftb abtdkstb zb wgczkdw » en version chiffrée. Dans le texte chiffré, la lettre *b* est la plus fréquente et nous pouvons deviner, à raison, que la lettre correspondante dans le texte en clair est le *e* (la lettre la plus fréquente dans un texte en français). Ensuite, nous pouvons examiner les mots de deux lettres, comme *jb* dans le texte chiffré et deviner que le caractère en clair qui correspond au *j* doit être *c*, *d*, *j*, *l*, *m*, *n*, *s* ou *t*, puisque les mots de deux lettres qui se terminent par *e* sont *ce*, *de*, *je*, *le*, *me*, *ne*, *se* et *te*. En procédant de la sorte, nous pouvons arriver à la clé de chiffrement.

Si nous chiffrons des numéros de cartes bancaires, nous n'avons pas à nous préoccuper de la fréquence des lettres ni de leurs combinaisons. En revanche, les dix chiffres ne donnent que 10! façons différentes de convertir un chiffre en un autre, c'est-à-dire seulement 3 628 800. Pour un ordinateur, c'est très peu, notamment en comparaison des $10^{16}$ numéros de carte bancaire possibles (seize chiffres décimaux). L'espion peut automatiser les tentatives d'achat de produits avec chacune des 10! possibilités, avec des chances de succès pour d'autres numéros de cartes bancaires que le nôtre.

Vous aurez peut-être remarqué un autre problème avec le chiffrement par simple substitution : l'expéditeur et le destinataire doivent se mettre d'accord sur la clé. Par ailleurs, si nous envoyons des messages différents à des tiers différents et si nous ne voulons pas que chaque tiers puisse déchiffrer le message destiné aux autres personnes, nous devons définir une clé distincte pour chacun.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

# Cryptographie à clé symétrique

Lorsque l'expéditeur et le destinataire utilisent la même clé, ils mettent en œuvre une **cryptographie à clé symétrique**. D'une manière ou d'une autre, ils doivent au préalable se mettre d'accord sur la clé à utiliser.

## Masques jetables

En supposant pour le moment que nous ne sommes pas prêts à utiliser la cryptographie à clé symétrique, mais qu'un chiffrement par simple substitution n'est pas suffisamment sûr, une autre solution est apportée par les masques jetables. Le masque jetable opère au niveau des bits. Comme vous le savez, un *bit* ne peut prendre que deux valeurs : 0 ou 1. Les ordinateurs numériques mémorisent les informations sous forme d'une suite de bits. Certaines suites représentent des nombres, d'autres des caractères (en utilisant des jeux de caractères ASCII ou Unicode), et d'autres encore, des instructions exécutées par l'ordinateur.

Les masques jetables appliquent un **OU exclusif** (XOR) entre des bits. Nous employons le symbole $\oplus$ pour représenter cette opération :

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

Voici le fonctionnement du OU exclusif : si $x$ est un bit, alors $x \oplus 0$ donne $x$, et $x \oplus 1$ donne l'opposé de $x$. Par ailleurs, si $x$ et $y$ sont des bits, alors $(x \oplus y) \oplus y = x$. Autrement dit, un OU exclusif entre $x$ et deux fois la même valeur donne $x$.

Supposons que nous souhaitions envoyer un message d'un bit. Le message chiffré transmis peut donc être un 0 ou un 1. Nous devons nous mettre d'accord avec le destinataire pour savoir si la valeur envoyée est celle que nous voulons envoyer ou son opposé. Du point de vue de l'opération, nous devons savoir si le OU exclusif se fait entre le bit et 0 ou 1. Si le destinataire applique ensuite au bit chiffré reçu un OU exclusif avec la valeur que nous avons appliquée, c'est-à-dire la clé, il retrouve le texte d'origine en clair.

Supposons à présent que nous voulions envoyer un message de deux bits. Nous pouvons les laisser tels quels, nous pouvons les inverser, nous pouvons inverser le premier mais pas le second, ou nous pouvons inverser le second mais pas le premier. À nouveau, nous devons nous mettre d'accord avec le destinataire pour savoir quels bits sont inversés. Du point de vue du OU exclusif sur deux bits, nous devons savoir quelle suite de deux bits, entre 00, 01, 10 et 11, représente la clé utilisée pour créer le texte chiffré. De nouveau, le destinataire peut appliquer un OU exclusif sur le texte

Copyright © 2013 Dunod.

chiffré de deux bits avec la même clé de deux bits pour retrouver le texte d'origine en clair.

Si le texte en clair occupe $b$ bits – il correspond peut-être à des caractères ASCII ou Unicode qui totalisent $b$ bits –, alors nous pouvons générer une clé sous forme d'une séquence aléatoire de $b$ bits, transmettre au destinataire les $b$ bits de la clé, et ensuite effectuer un OU exclusif bit à bit entre le texte en clair et la clé pour obtenir le texte chiffré. Lorsque le destinataire reçoit le texte chiffré sur $b$ bits, il applique un OU exclusif bit à bit entre ce texte chiffré et la clé de manière à retrouver le texte en clair sur $b$ bits. Ce système est appelé *masque jetable*[1], la clé étant le **masque**.

Tant que les bits de la clé sont choisis aléatoirement (nous y reviendrons plus loin), il est presque impossible pour un espion de décrypter le texte chiffré en devinant la clé. Même s'il a des informations sur le texte en clair, par exemple qu'il est rédigé en français, pour n'importe quel texte chiffré et n'importe quel texte en clair potentiel, il existe une clé qui permet de passer de l'un à l'autre[2], et cette clé est le OU exclusif bit à bit entre le texte en clair potentiel et le texte chiffré. (En effet, si le texte en clair potentiel est $t$, le texte chiffré est $c$ et la clé est $k$, alors non seulement $t \oplus k = c$, mais $t \oplus c = k$ ; l'opération $\oplus$ s'applique bit à bit à $t$, $k$ et $c$, et un OU exclusif entre $i$-ième bit de $t$ et le $i$-ième bit de $k$ est égal au $i$-ième bit de $c$.) Un chiffrement avec un masque jetable évite donc que cet espion n'obtienne des informations supplémentaires sur le texte en clair.

Les masques jetables permettent d'obtenir une sécurité de bon niveau, mais les clés ont autant de bits que le texte en clair, ces bits doivent être choisis de manière aléatoire et les clés doivent être connues des tiers à l'avance. Comme son nom l'implique, un masque jetable ne doit être utilisé qu'une seule fois. Si nous utilisons la même clé $k$ pour les textes en clair $t_1$ et $t_2$, alors $(t_1 \oplus k) \oplus (t_2 \oplus k) = t_1 \oplus t_2$, ce qui permet de savoir où les deux textes en clair ont les mêmes bits.

---

1. Ce nom (*one-time pad* en anglais) vient de la mise en œuvre de cette idée à une époque antérieure à l'arrivée de l'informatique, où chaque partie disposait d'un bloc (*pad*) de papier avec une clé écrite sur chaque feuille. Chaque partie disposait de la même séquence de clés. Une clé ne servait qu'une seule fois. La feuille correspondante était ensuite enlevée du bloc et la clé suivante était révélée. Ce système papier utilisait un chiffrement par décalage, mais sur la base des lettres, avec chaque lettre de la clé donnant le décalage à effectuer, de 0 pour *a* à 25 pour *z*. Par exemple, puisque *z* signifie un décalage de 25, *m* un décalage de 12, et *n* un décalage de 13, la clé *zmn* convertit le texte en clair « dog » en texte chiffré « cat ». Contrairement à un système fondé sur le OU exclusif, le décalage des lettres du texte chiffré dans le même sens et avec la même clé ne permet pas de revenir au texte en clair ; dans notre exemple, nous obtenons « bmg ». À la place, le décalage des lettres du texte chiffré doit se faire dans le sens opposé.
2. Avec le fonctionnement lettre par lettre de la note précédente, la clé *zmn* convertit le texte en clair « dog » en texte chiffré « cat », mais nous pouvons obtenir ce texte chiffré avec un texte en clair différent, « elk », et une autre clé, *ypj*.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

## Chiffrement par blocs et chaînage

Lorsque la longueur du texte en clair est importante, le chiffrement par masque jetable impose un masque également long, ce qui peut être ennuyeux. À la place, certains systèmes à clé symétrique associent deux techniques supplémentaires : ils prennent une clé plus courte, décomposent le texte en clair en plusieurs blocs et appliquent la clé à chaque bloc. Autrement dit, ils considèrent que le texte en clair est constitué de $l$ blocs, $t_1, t_2, t_3,\ldots, t_l$, et ils procèdent au chiffrement de ces blocs de texte en clair de façon à obtenir $l$ blocs de texte chiffré, $c_1, c_2, c_3,\ldots, c_l$. Un tel système est un **chiffrement par blocs**.

Dans la pratique, le chiffrement par blocs se fait à l'aide d'un système un peu plus compliqué qu'un simple OU exclusif avec le masque jetable. L'un des cryptosystèmes à clé symétrique très répandu, AES (*Advanced Encryption Standard*), comprend un chiffrement par blocs. Nous n'entrerons pas dans les détails d'AES, hormis pour dire qu'il met en œuvre des méthodes élaborées de découpage du texte en clair pour générer le texte chiffré. Il utilise une clé dont la taille peut être de 128, 192 ou 256 bits, ainsi que des blocs de 128 bits.

Toutefois, le chiffrement par blocs pose un problème. Si le même bloc apparaît deux fois dans le texte en clair, alors le même bloc chiffré apparaîtra deux fois dans le texte chiffré. Pour résoudre ce problème, une solution se fonde sur le **chaînage des blocs de chiffrement**. Supposons qu'une personne souhaite nous envoyer un message chiffré. Elle découpe le texte en clair $t$ en $l$ blocs $t_1, t_2, t_3,\ldots, t_l$, et elle crée les $l$ blocs $c_1, c_2, c_3,\ldots, c_l$ du texte chiffré de la manière suivante. Supposons qu'un bloc soit chiffré en lui appliquant une fonction $E$ et que nous déchiffrions un bloc en lui appliquant une fonction $D$. La personne crée le premier bloc du texte chiffré, $c_1$, de façon attendue : $c_1 = E(t_1)$. En revanche, pour chiffrer le deuxième bloc, elle lui applique un OU exclusif bit à bit avec $c_1$, c'est-à-dire $c_2 = E(c_1 \oplus t_2)$. Pour le troisième bloc, le OU exclusif se fait avec $c_2$ : $c_3 = E(c_2 \oplus t_3)$. Et ainsi de suite. De façon générale, la génération du $i$-ième bloc du texte chiffré se fonde sur le $(i-1)$-ième bloc du texte chiffré et sur le $i$-ième bloc de texte en clair : $c_i = E(c_{i-1} \oplus t_i)$. Cette formule fonctionne également pour la génération de $c_1$ à partir de $t_1$ si $c_0$ contient uniquement des zéros (car $0 \oplus x$ donne $x$). Pour déchiffrer le texte, nous commençons par calculer $t_1 = D(c_1)$. À partir de $c_1$ et $c_2$, nous pouvons obtenir $t_2$ en commençant par calculer $D(c_2)$, qui est égal à $c_1 \oplus t_2$, puis en effectuant un OU exclusif entre le résultat et $c_1$. De façon générale, nous déchiffrons $c_i$ pour obtenir $t_i$ en calculant $t_i = D(c_i) \oplus c_{i-1}$. Comme pour le chiffrement, cette formule permet d'obtenir $t_1$ si nous commençons avec $c_0$ constitué intégralement de zéros.

Pourtant tout n'est pas gagné. Même avec un chaînage des blocs, si l'expéditeur nous envoie deux fois le même message, nous recevons à chaque

Copyright © 2013 Dunod.

fois la même suite de blocs de texte chiffré. Un espion peut alors savoir que nous recevons deux fois le même message, ce qui peut lui fournir des informations intéressantes. Une solution consiste à ne pas partir avec $c_0$ constitué uniquement de zéros. À la place, il est généré de façon aléatoire et sert au chiffrement du premier bloc de texte en clair. Nous l'utilisons également pour le déchiffrement du premier bloc du texte chiffré. Un tel $c_0$ généré aléatoirement est appelé *vecteur d'initialisation*.

### S'accorder sur les informations communes

Pour que la cryptographie à clé symétrique fonctionne, l'expéditeur et le destinataire doivent se mettre d'accord sur la clé. Par ailleurs, s'ils utilisent un chiffrement par blocs avec chaînage des blocs, ils doivent disposer du même vecteur d'initialisation. Comme vous pouvez l'imaginer, il est peu pratique de s'accorder sur ces valeurs au préalable. Comment l'expéditeur et le destinataire peuvent-ils procéder ? Nous vous y reviendrons plus loin dans ce chapitre, à la page 159, en présentant un cryptosystème hybride capable de les transmettre de façon sécurisée.

## Cryptographie à clé publique

Pour que le destinataire d'un message chiffré soit capable de le déchiffrer, il est évident que l'expéditeur et le destinataire doivent tous deux connaître la clé qui a été utilisée pour le chiffrement. Eh bien c'est faux !

Avec la **cryptographie à clé publique**, ou **cryptographie asymétrique**, chaque partie possède deux clés : une **clé publique** et une **clé privée** (secrète). Nous allons décrire ce fonctionnement avec deux parties, vous et moi. Ma clé publique sera notée $P$ et ma clé privée, $S$. Vous aurez vos propres clés publique et privée. Les autres tiers qui pourraient participer auraient également leurs propres clés publique et privée.

Les clés privées doivent rester secrètes, contrairement aux clés publiques que tout le monde peut connaître. Elles peuvent même être placées dans un répertoire centralisé, de sorte que tout le monde ait accès à la clé publique des autres. Dans les conditions appropriées, vous et moi pouvons utiliser ces clés pour le chiffrement et le déchiffrement. Par « conditions appropriées », j'entends qu'il existe des fonctions qui utilisent les clés publique et privée pour chiffrer du texte en clair et pour déchiffrer du texte chiffré. Notons $F_P$ la fonction que j'utilise avec ma clé publique, et $F_S$ celle que j'utilise avec ma clé privée.

Les clés publique et secrète ont une relation particulière :

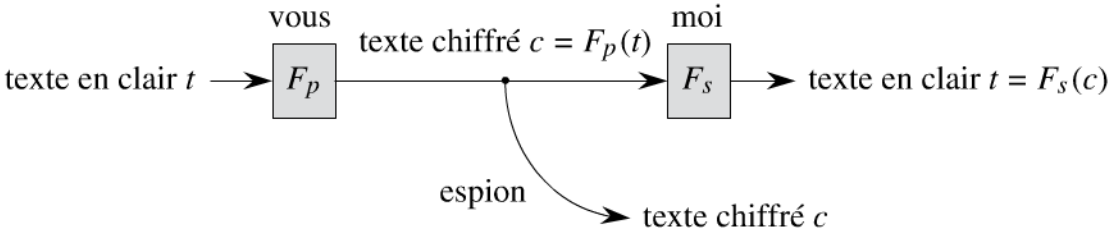
$$t = F_S(F_P(t))$$

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

Par conséquent, si vous utilisez ma clé publique pour chiffrer le texte en clair et si j'utilise ensuite ma clé privée pour déchiffrer le texte chiffré reçu, je retrouve le texte en clair d'origine. Certaines applications de la cryptographie à clé publique exigent que $t = F_P(F_S(t))$. Dans ce cas, si je chiffre du texte en clair avec ma clé secrète, n'importe qui peut déchiffrer le résultat.

Tout le monde doit pouvoir exécuter rapidement ma fonction $F_P$ de clé publique, mais je dois être le seul à pouvoir exécuter ma fonction $F_S$ de clé privée en un temps raisonnable. Le temps nécessaire pour deviner ma $F_S$ sans connaître ma clé privée doit être extrêmement long pour toutes les autres personnes. (Je suis volontairement vague ici, mais nous verrons bientôt une implémentation réelle de la cryptographie à clé publique.) Les mêmes conditions s'appliquent aux clés publique et privée des autres personnes : la fonction $F_P$ de clé publique opère rapidement, mais seul le détenteur de la clé privée peut exécuter en un temps raisonnable la fonction $F_S$ de clé privée.

Voici comment se déroule l'envoi d'un message de vous à moi en utilisant la cryptographie à clé publique :




Vous partez du texte en clair $t$. Vous récupérez ma clé publique $P$ ; je vous la fournis ou vous la trouvez dans un répertoire. À l'aide de la clé $P$, vous chiffrez le texte en clair et obtenez le texte chiffré $c = F_P(t)$, ce qui prend peu de temps. Vous m'envoyez le texte chiffré. Un espion qui intercepterait ce que vous envoyez verrait uniquement le texte chiffré. Je prends le texte chiffré $c$ et le déchiffre à l'aide de ma clé privée. J'obtiens le texte en clair $t = F_S(c)$. Vous, ou une autre personne, pouvez effectuer le chiffrement de manière relativement rapide, mais je suis le seul à pouvoir déchiffrer le texte reçu et obtenir le texte en clair en un temps raisonnable.

Dans la pratique, nous devons nous assurer que les fonctions $F_P$ et $F_S$ se complètent correctement. Nous voulons que $F_P$ génère un texte chiffré différent pour chaque texte en clair possible. Supposons à la place que $F_P$ donne le même résultat pour deux textes en clair différents, $t_1$ et $t_2$ ; autrement dit, $F_P(t_1) = F_P(t_2)$. Lorsque je reçois un texte chiffré $F_P(t1)$ et tente de le déchiffrer en le passant à $F_S$, je ne sais pas si j'obtiens $t_1$ ou $t_2$. Cela dit, il n'est pas plus mal que le chiffrement comporte un élément aléatoire, de sorte que le même texte en clair donne des textes chiffrés différents chaque fois qu'il est traité par $F_P$. (Le cryptosystème RSA, que

Copyright © 2013 Dunod.

nous décrivons ci-après, est beaucoup plus sûr lorsque le texte en clair correspond uniquement à une petite partie de ce qui a été chiffré ; le gros de l'information chiffrée étant alors du remplissage aléatoire.) Bien entendu, la fonction $F_S$ de déchiffrement doit être conçue pour tenir compte de cela, de sorte qu'elle ne puisse pas obtenir le même texte en clair à partir de différents textes chiffrés[1].

Il existe toutefois un problème. Le texte en clair $t$ peut prendre un nombre quelconque de valeurs (en réalité, il peut être arbitrairement long) et le nombre de valeurs du texte chiffré dans lesquelles $P$ va convertir $t$ doit être au moins égal au nombre de valeurs possibles pour $t$. Comment pouvons-nous construire les fonctions $F_P$ et $F_S$ en tenant compte des contraintes de facilité de calcul de $F_P$ pour tout le monde et de facilité de calcul de $F_S$ uniquement pour moi ? La méthode est complexe, mais elle est possible si nous pouvons limiter le texte en clair, c'est-à-dire si nous utilisons un chiffrement par blocs.

## Cryptosystème RSA

La cryptographie à clé publique est un concept intéressant, mais elle se fonde sur la possibilité de trouver des fonctions $F_P$ et $F_S$ qui se complètent parfaitement. $F_P$ doit être facile à calculer par tout le monde, $F_S$ n'est simple à calculer que pour le détenteur de la clé privée. Un système qui satisfait ces deux critères est un **cryptosystème à clé publique** et le **cryptosystème RSA**[2], ou simplement **RSA**, en fait partie.

RSA se fonde sur plusieurs aspects de la théorie des nombres, dont plusieurs ont un rapport avec l'**arithmétique modulaire**. En arithmétique modulaire, nous choisissons un entier strictement positif, $n$, et, dès que nous arrivons à $n$, nous revenons immédiatement à 0. Elle est comparable à l'arithmétique normale avec les entiers, mais nous divisons toujours par $n$ et prenons le reste. Par exemple, si nous opérons en modulo 5, les seules valeurs possibles sont alors 0, 1, 2, 3 et 4, et $3 + 4 = 2$ car 2 est le reste de la division de 7 par 5. Définissons l'opérateur *mod* qui permet de calculer

1. Le base-ball se fonde sur un système comparable. Les dirigeants et les coaches indiquent aux joueurs leur mode de jeu en utilisant un système élaboré de gestes appelés « signes ». Par exemple, toucher l'épaule droite peut indiquer une action de type « frappe-et-court », tandis que toucher l'épaule gauche peut correspondre à une action de type « amorti ». Un manager ou un coach effectue une longue suite de signes, mais seuls certains ont un sens ; les autres ne sont que des leurres. Celui qui fait les signes et celui qui les regarde ont mis en place un système qui leur permet de s'accorder sur les signes significatifs. Cela peut dépendre de l'ordre de la suite des signes, ou d'un signe qui sert d'indicateur. Le manager ou le coach peut produire une longue suite arbitraire de signes afin d'indiquer un mode de jeu particulier, mais la plupart d'entre eux n'ont aucune importance.
2. Le nom vient de ses inventeurs, Ronald Rivest, Adi Shamir et Leonard Adelman.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

le reste, de sorte que nous puissions écrire $7 \; mod \; 5 = 2$. L'arithmétique modulaire équivaut à l'arithmétique de l'horloge, mais en remplaçant 12 par 0 sur le cadran. Si nous nous couchons à 11 et dormons pendant huit heures, nous nous réveillons à 7 : $(11 + 8) \; mod \; 12 = 7$.

Le point intéressant de l'arithmétique modulaire vient du fait que nous pouvons placer des opérations *mod* au milieu d'une expression sans changer le résultat :[1]

$$(a + b) \; mod \; n = ((a \; mod \; n) + (b \; mod \; n)) \; mod \; n$$
$$ab \; mod \; n = ((a \; mod \; n)(b \; mod \; n)) \; mod \; n$$
$$a^b \; mod \; n = (a \; mod \; n)^b \; mod \; n$$

Pour tout entier $x$, nous avons également $xn \; mod \; n$ égal à 0.

Par ailleurs, pour que RSA remplisse les critères d'un cryptosystème à clé publique, deux propriétés liées aux nombres premiers doivent être vérifiées. Vous le savez certainement, un **nombre premier** est un entier supérieur à 1 qui possède uniquement deux diviseurs entiers : 1 et lui-même. Par exemple, 7 est un nombre premier, contrairement à 6 qui est divisible par 2 et par 3. La première propriété sur laquelle se fonde RSA est la suivante : si nous avons un nombre qui est le produit de deux grands nombres premiers secrets, alors personne ne peut déterminer ces facteurs en un temps raisonnable. Au Chapitre 1, nous avons expliqué qu'il était possible de tester tous les diviseurs impairs jusqu'à la racine carrée d'un nombre, mais que si ce nombre était vraiment grand – des centaines voire des milliers de chiffres –, alors sa racine carrée comprenait un nombre de chiffres divisé par deux, ce qui pouvait rester extrêmement grand. Bien qu'il soit théoriquement possible de trouver ces facteurs, les ressources nécessaires (temps et/ou puissance de calcul) rendent cette recherche irréalisable[2].

Voici la seconde propriété : même s'il est difficile de factoriser un grand nombre premier, il n'est pas compliqué de déterminer si un grand nombre est premier. Vous pourriez penser qu'il est impossible de déterminer qu'un nombre n'est pas premier, autrement dit qu'il est *composé*, sans trouver

Copyright © 2013 Dunod.

1. Afin d'illustrer cela, voyons si $ab \; mod \; n = ((a \; mod \; n)(b \; mod \; n)) \; mod \; n$. Supposons que $a \; mod \; n = x$ et que $b \; mod \; n = y$. Il existe alors des entiers $i$ et $j$ tels que $a = ni + x$ et $b = nj + y$, et donc que :

$$\begin{aligned} ab \; mod \; n &= (ni + x)(nj + y) mod \; n \\ &= (n^2ij + xnj + yni + xy) \; mod \; n \\ &= \big((n^2ij \; mod \; n) + (xnj \; mod \; n) + (yni \; mod \; n) + (xy \; mod \; n)\big) \; mod \; n \\ &= xy \; mod \; n \\ &= ((a \; mod \; n)(b \; mod \; n)) \; mod \; n \end{aligned}$$

2. Par exemple, si le nombre est constitué de 1 000 chiffres, alors sa racine carrée comprend 500 chiffres et peut aller jusqu'à la valeur $2^{500}$. Même s'il était possible de tester un milliard de milliards de diviseurs par seconde, il y a longtemps que le soleil aurait disparu avant d'atteindre $2^{500}$.

au moins un diviseur non trivial, c'est-à-dire différent de 1 ou du nombre lui-même. Vous vous trompez. Une manière de procéder est le test de primalité AKS[1]. Il s'agit du premier algorithme capable de déterminer si un nombre de $n$ bits est premier en un temps $O(n^c)$ pour $c$ constant. Bien que cet algorithme soit considéré efficace en théorie, il n'est pas encore envisageable pour les grands nombres. À la place, nous pouvons employer le test de primalité de Miller-Rabin, mais il a pour inconvénient de parfois se tromper, en déclarant qu'un nombre composé est premier. (En revanche, s'il déclare qu'un nombre est composé, alors il l'est réellement.) La bonne nouvelle est que son taux d'erreur est de 1 sur $2^s$, avec $s$ étant n'importe quelle valeur strictement positive que nous souhaitons. Par conséquent, si nous acceptons une erreur tous les $2^{50}$ tests, nous pouvons déterminer avec une quasi-certitude si un nombre est premier. Au Chapitre 1, nous avons indiqué que $2^{50}$ est environ un million de milliards, c'est-à-dire environ 1 000 000 000 000 000. Si une erreur sur $2^{50}$ tests est encore trop fréquente, avec un petit effort supplémentaire, nous pouvons arriver à une erreur sur $2^{60}$ tests ; $2^{60}$ est environ 1 000 fois supérieur à $2^{50}$. En effet, puisque le temps d'exécution du test de Miller-Rabin croît de façon linéaire avec le paramètre $s$, augmenter celui-ci de 10, c'est-à-dire de 50 à 60, augmente le temps d'exécution de seulement 20 %, mais diminue le taux d'erreur d'un facteur $2^{10}$, c'est-à-dire 1 024.

Voici comment nous préparons la mise en place du cryptosystème RSA. Après que nous aurons donné son fonctionnement, nous aborderons quelques détails.

1. Choisir de façon aléatoire deux très grands nombres premiers, $p$ et $q$, qui ne sont pas égaux. Ils doivent être constitués d'au moins 1 024 bits, c'est-à-dire d'au moins 309 chiffres décimaux. Plus ils sont grands, mieux c'est.
2. Calculer $n = pq$. Ce nombre comprend au moins 2 048 bits, c'est-à-dire au moins 618 chiffres décimaux.
3. Calculer $r = (p-1)(q-1)$, qui est presque aussi grand que $n$.
4. Choisir un petit entier impair $e$ qui est **premier relativement** à $r$ : le seul diviseur commun entre $e$ et $r$ doit être 1. Tout petit entier qui respecte cette contrainte fera l'affaire.
5. Calculer $d$ en tant qu'**inverse multiplicatif** de $e$, modulo $r$. Autrement dit, $ed \bmod r$ doit être égal à 1.
6. Prendre le couple $P = (e,n)$ en tant que **clé publique RSA**.
7. Prendre le couple $S = (d,n)$ en tant que **clé privée RSA** et ne le donner à personne.

1. D'après le nom de ses inventeurs, Manindra Agrawal, Neeraj Kayal et Nitin Saxena.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

8. Définir les fonctions $F_P$ et $F_S$ suivantes :

$$F_P(x) = x^e \ mod \ n$$
$$F_S(x) = x^d \ mod \ n$$

Ces fonctions peuvent opérer sur un bloc de texte en clair ou un bloc de texte chiffré, dont les bits que nous interprétons représentent de grands entiers.

Prenons un exemple, mais en utilisant de petites valeurs afin de comprendre ce qui se passe.

1. Choisir les nombres premiers $p = 17$ et $q = 29$.
2. Calculer $n = pq = 493$.
3. Calculer $r = (p - 1)(q - 1) = 448$.
4. Choisir $e = 5$, qui est premier relativement à 448.
5. Calculer $d = 269$. En effet, $ed = 5 \cdot 269 = 1\ 345$, et donc $ed \ mod \ r = 1\ 345 \ mod \ 448 = (3 \cdot 448) + 1 \ mod \ 448 = 1$.
6. Prendre le couple $P = (5,493)$ en tant que clé publique RSA.
7. Prendre le couple $S = (269,493)$ en tant que clé privée RSA.
8. Comme exemple, calculons $F_P(327)$ :

$$\begin{aligned} F_P(327) &= 327^5 \ mod \ 493 \\ &= 3\ 738\ 856\ 210\ 407 \ mod \ 493 \\ &= 259 \end{aligned}$$

Si nous calculons $F_S(259) = 259^{269} \ mod \ 493$, nous devons obtenir 327. C'est le cas ; je vais vous épargner la lecture de tous les chiffres de $259^{269}$. Vous pouvez rechercher sur Internet une calculatrice à précision quelconque et vérifier le résultat (je l'ai fait). Puisque nous opérons en arithmétique modulaire, nous n'avons pas besoin de calculer la valeur réelle de $259^{269}$. Nous pouvons exprimer tous les résultats intermédiaires modulo 493 et donc démarrer avec le produit 1 et effectuer 269 fois la chose suivante : multiplier ce que nous obtenons par 259 et prendre le produit modulo 493. Nous arrivons à 327. (Je l'ai fait, enfin je l'ai fait faire à un programme que j'ai écrit.)

Voici les détails à prendre en compte dans la mise en place et l'utilisation de RSA :

- Comment pouvons-nous manipuler des nombres constitués de centaines de chiffres ?
- Bien que vérifier la primalité d'un nombre ne constitue pas un obstacle, comment pouvons-nous savoir que nous trouverons de grands nombres premiers en un temps raisonnable ?
- Comment pouvons-nous trouver $e$ tel que $e$ et $r$ soient premiers entre eux ?

Copyright © 2013 Dunod.

- Comment pouvons-nous calculer $d$ tel qu'il soit l'inverse multiplicatif de $e$, modulo $r$ ?
- Si $d$ est grand, comment pouvons-nous calculer $x^d \ mod \ n$ en un temps raisonnable ?
- Comment pouvons-nous savoir que les fonctions $F_P$ et $F_S$ sont des inverses l'une de l'autre ?

## Arithmétique avec des grands nombres

Les nombres requis par RSA sont trop grands pour tenir dans les registres disponibles sur la plupart des ordinateurs, car leur taille est limitée à 64 bits. Heureusement, il existe des paquetages logiciels et même des langages de programmation, comme Python, qui permettent de manipuler des entiers de taille non figée.

Par ailleurs, RSA utilise une arithmétique modulaire, ce qui nous permet de limiter la taille des entiers calculés. Par exemple, pour le calcul de $x^d \ mod \ n$, nous calculerons les résultats intermédiaires de $x$ à différentes puissances mais tous modulo $n$. Autrement dit, tous les résultats intermédiaires se trouveront dans la plage 0 à $n - 1$. De plus, si nous fixons les tailles maximales de $p$ et de $q$, alors nous fixons la taille maximale de $n$, ce qui signifie qu'il est possible d'implémenter RSA avec un matériel spécialisé.

## Rechercher un grand nombre premier

Nous pouvons rechercher un grand nombre premier en répétant la génération aléatoire d'un grand nombre impair et en utilisant le test de Miller-Rabin pour vérifier s'il est premier, puis en nous arrêtant dès que c'est le cas. Cette approche suppose qu'un grand nombre premier sera trouvé en peu de temps. Mais, que faire si les grands nombres premiers sont extrêmement rares ? Nous pourrions passer beaucoup de temps à rechercher une aiguille dans une meule de foin (ou, dans ce cas, de nombres composés).

Il est inutile de s'inquiéter, car le **théorème des nombres premiers** nous indique que plus $m$ tend vers l'infini, plus le nombre de nombres premiers inférieurs ou égaux à $m$ tend vers $m$ / $\ln m$, avec $\ln m$ le logarithme népérien de $m$. Si nous choisissons un entier $m$ aléatoire, il y a environ une chance sur $\ln m$ pour qu'il soit premier. La théorie des probabilités nous indique que nous devons, en moyenne, tester uniquement $\ln m$ nombres proches de $m$ avant d'en trouver un qui soit premier. Si nous recherchons des nombres premiers $p$ et $q$ de 1 024 bits, alors $m$ est égal à $2^{1\,024}$ et $\ln m$ vaut environ 710. Un ordinateur est tout à fait capable d'effectuer rapidement le test de primalité de Miller-Rabin sur 710 nombres.

En pratique, il existe un test de primalité plus simple que celui de Miller-Rabin. Le **petit théorème de Fermat** stipule que si $m$ est un nombre premier,

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

alors $x^{m-1} \bmod m$ est égal à 1 pour tout nombre $x$ dans la plage 1 à $m-1$. Le contraire – si $x^{m-1} \bmod m$ est égal à 1 pour tout nombre $x$ dans la plage 1 à $m-1$ alors $m$ est premier – n'est pas nécessairement vrai, mais les exceptions sont très rares avec les très grands nombres. En réalité, il est presque toujours suffisant de tester des entiers impairs $m$ et de déclarer que $m$ est premier si $2^{m-1} \bmod m$ est égal à 1. Nous verrons à la page 158 comment calculer $2^{m-1} \bmod m$ avec seulement $\Theta(\lg m)$ multiplications.

## Rechercher deux nombres premiers entre eux

Nous devons trouver un petit entier impair $e$ qui soit premier relativement à $r$. Deux nombres sont premiers entre eux s'ils ont 1 pour plus grand commun diviseur. Pour calculer le plus grand commun diviseur de deux entiers, nous allons employer un algorithme qui date de l'époque du mathématicien grec Euclide. Un théorème de la théorie des nombres stipule que si $a$ et $b$ sont deux entiers différents de zéro, leur plus grand commun diviseur $g$ est égal à $ai + bj$ avec $i$ et $j$ entiers. (De plus, $g$ est le plus petit nombre qui peut être formé de cette manière, mais cela ne nous intéresse pas.) L'un des coefficients $i$ ou $j$ peut être négatif. Par exemple, le plus grand commun diviseur de 30 et de 18 est 6, et $6 = 30i + 18j$ pour $i = -1$ et $j = 2$.

L'algorithme d'Euclide est donné ci-après, sous une forme qui permet d'obtenir le plus grand commun diviseur $g$ pour les nombres $a$ et $b$, ainsi que les coefficients $i$ et $j$. Ces coefficients nous seront utiles ultérieurement, lorsque nous devrons trouver l'inverse multiplicatif de $e$, modulo $r$. Si nous disposons d'une valeur potentielle pour $e$, nous appelons EUCLIDE$(r,e)$. Si le premier élément du triplet retourné par la procédure est 1, la valeur potentielle pour $e$ est première relativement à $r$. S'il s'agit d'un autre nombre, alors $r$ et la valeur potentielle pour $e$ ont en commun un diviseur supérieur à 1 et ces deux nombres ne sont donc pas premiers entre eux.

*Procédure* EUCLIDE$(a,b)$

*Entrées* : $a$ et $b$ : deux entiers.

*Sortie* : un triplet $(g,i,j)$ tel que $g$ est le plus grand commun diviseur de $a$ et de $b$, avec $g = ai + bj$.

1. Si $b$ est égal à 0, alors retourner le triplet $(a,1,0)$.
2. Sinon ($b$ est différent de 0), faire la chose suivante :
   A. Appeler récursivement EUCLIDE$(b, a \bmod b)$, et affecter le résultat obtenu au triplet $(g,i',j')$. Autrement dit, affecter $g$ au premier élément du triplet retourné, $i'$ au deuxième élément et $j'$ au troisième élément.
   B. Fixer $i$ à $j'$.

Copyright © 2013 Dunod.

C. Fixer $j$ à $\lfloor i' - a / b \; j' \rfloor$.
D. Retourner le triplet $(g,i,j)$.

Je n'expliquerai pas pourquoi cette procédure fonctionne [1], ni n'analyserai son temps d'exécution. Je dirai simplement que si nous appelons EUCLIDE $(r,e)$, alors le nombre d'appels récursifs est $O(\lg e)$. Nous pouvons donc vérifier rapidement si 1 est le plus grand commun diviseur de $r$ et d'une valeur potentielle pour $e$. (N'oubliez pas que $e$ est petit.) Si ce n'est pas le cas, nous essayons une autre valeur pour $e$, jusqu'à en trouver une qui soit première relativement à $r$. Combien de candidats doit-on s'attendre à tester ? Peu. Si nous limitons nos choix pour $e$ aux nombres premiers impairs inférieurs à $r$ (une vérification facile à réaliser avec le test de Miller-Rabin ou du petit théorème de Fermat), n'importe quelle valeur aura de grande chance d'être un nombre premier relativement à $r$. En effet, en raison du théorème des nombres premiers, il existe environ $r / \ln r$ nombres premiers inférieurs à $r$, mais un autre théorème montre que $r$ ne peut pas avoir plus de $\lg r$ facteurs premiers. Par conséquent, nous avons peu de risque de tomber sur un facteur premier de $r$.

## Calculer des inverses multiplicatifs en arithmétique modulaire

Dès que nous avons $r$ et $e$, nous devons calculer $d$ comme l'inverse multiplicatif de $e$, modulo $r$, de sorte que $ed \bmod r$ soit égal à 1. Nous savons déjà que l'appel EUCLIDE$(r,e)$ a retourné un triplet de la forme $(1,i,j)$, que 1 est le plus grand commun diviseur de $e$ et de $r$ (car ils sont premiers entre eux), et que $1 = ri + ej$. Nous pouvons simplement fixer $d$ à $j \bmod r$[2] En effet, nous travaillons modulo $r$ et nous pouvons donc prendre les deux côtés modulo $r$ :

$$\begin{aligned} 1 \bmod r &= (ri + ej) \bmod r \\ &= ri \bmod r + ej \bmod r \\ &= 0 + ej \bmod r \\ &= ej \bmod r \end{aligned}$$

1. Puisque l'appel EUCLIDE$(0,0)$ retourne le triplet $(0,1,0)$, 0 est considéré comme le plus grand commun diviseur de 0 et de 0. Cela pourra vous choquer. Toutefois, puisque $r$ est strictement positif, le paramètre $a$ du premier appel à EUCLIDE sera strictement positif et, dans tous les appels récursifs, $a$ restera strictement positif. Par conséquent, peu importe le triplet retourné par EUCLIDE$(0,0)$.
2. Rappelons que $j$ peut être négatif. Une manière de calculer $j \bmod r$ lorsque $j$ est négatif et $r$ est positif consiste à démarrer avec $j$ et à constamment lui ajouter $r$ jusqu'à obtenir un nombre positif. Ce nombre est égal à $j \bmod r$. Par exemple, dans le calcul de $-27 \bmod 10$, nous obtenons les nombres -27, -17, -7 et 3. Dès que nous arrivons à 3, nous nous arrêtons et concluons que $-27 \bmod 10$ est égal 3.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

$$= (e \bmod r) \cdot (j \bmod r) \bmod r$$
$$= e(j \bmod r) \bmod r$$

(La dernière ligne vient du fait que $e < r$, ce qui implique que $e \bmod r = e$.) Nous avons donc $1 = e(j \bmod r) \bmod r$. Autrement dit, nous pouvons fixer $d$ à la valeur de $j$ donnée par le triplet retourné par l'appel EUCLIDE$(r,e)$, pris modulo $r$. Nous prenons $j \bmod r$ plutôt que juste $j$ car $j$ ne se trouve pas obligatoirement dans la plage 0 à $r-1$.

## Élever rapidement un nombre à une puissance entière

Même si $e$ est petit, $d$ peut être grand, et nous devons calculer $x^d \bmod n$ de façon à calculer la fonction $F_S$. Bien que nous puissions travailler modulo $n$, autrement dit avec des valeurs dans la plage 0 à $n-1$, nous ne souhaitons pas multiplier des nombres à $d$ reprises. Heureusement, ce n'est pas obligatoire. Nous pouvons multiplier des nombres à $\Theta(\lg d)$ reprises uniquement, en utilisant une technique d'*exponentiation rapide*. Elle peut également servir pour le test de primalité fondé sur le petit théorème de Fermat.

Le principe est le suivant. Nous savons que $d$ est positif. Supposons tout d'abord que $d$ soit pair. Alors $x^d$ est égal à $\left(x^{d/2}\right)^2$. Supposons à présent que $d$ soit impair. Alors $x^d$ est égal à $\left(x^{(d-1)/2}\right)^2 \cdot x$. Ces observations conduisent à une solution récursive pour le calcul de $x^d$, avec pour cas de base $d$ égal à 0 : $x^0$ est égal à 1. La procédure suivante se fonde sur cette approche, en effectuant une arithmétique modulo $n$ :

*Procédure* EXPONENTIATION-MODULAIRE$(x,d,n)$

*Entrées* : $x$, $d$, $n$ : trois entiers, avec $x$ et $d$ positifs, et $n$ strictement positif.

*Sortie* : la valeur de $x^d \bmod n$.

1. Si $d$ est égal à 0, alors retourner 1.
2. Sinon ($d$ est positif), si $d$ est pair, alors appeler récursivement EXPONENTIATION-MODULAIRE$(x,d/2,n)$, fixer $z$ au résultat de cet appel récursif et retourner $z^2 \bmod n$.
3. Sinon ($d$ est positif et impair), appeler récursivement EXPONENTIATION-MODULAIRE$(x,(d-1)/2,n)$, fixer $z$ au résultat de cet appel récursif et retourner $(z^2 \cdot x) \bmod n$.

Le paramètre $d$ est divisé par 2 à chaque appel récursif. Après au plus $\lfloor \lg d \rfloor + 1$ appels, $d$ arrive à 0 et la récursion se termine. Par conséquent, cette procédure multiplie des nombres à $\Theta(\lg d)$ reprises.

Copyright © 2013 Dunod.

## Montrer que les fonctions $F_p$ et $F_s$ sont des inverses

Les explications suivantes font la part belle à la théorie des nombres et à l'arithmétique modulaire. Si vous êtes prêt à accepter sans autre preuve que les fonctions $F_P$ et $F_S$ sont inverses l'une de l'autre, vous pouvez sauter les cinq prochains paragraphes et reprendre votre lecture à la section « Cryptosystèmes hybrides ».

Pour que RSA soit un cryptosystème à clé publique, les fonctions $F_P$ et $F_S$ doivent être inverses l'une de l'autre. Si nous prenons un bloc $t$ de texte en clair, le traitons comme un entier inférieur à $n$ et le passons à $F_P$, nous obtenons $t^e \mod n$. Si nous passons ce résultat à $F_S$, nous obtenons $(t^e)^d \mod n$, qui est égal à $t^{ed} \mod n$. Si nous inversons l'ordre d'utilisation de ces fonctions, d'abord $F_S$ puis $F_P$, nous obtenons $(t^d)^e \mod n$, qui est à nouveau égal à $t^{ed} \mod n$. Nous devons montrer que pour n'importe quel bloc $t$ de texte en clair, traité comme un entier inférieur à $n$, nous avons $t^{ed} \mod n$ égal à $t$.

Voici le principe général de notre approche. Rappelons que $n = pq$. Nous allons montrer que $t^{ed} \mod p = t \mod p$ et que $t^{ed} \mod q = t \mod q$. Ensuite, en utilisant un autre résultat de la théorie des nombres, nous conclurons que $t^{ed} \mod pq = t \mod pq$, autrement dit que $t^{ed} \mod n = t \mod n$, c'est-à-dire $t$ car $t$ est inférieur à $n$.

Nous devons à nouveau nous servir du petit théorème de Fermat et cela sera plus facile en expliquant pourquoi nous fixons $r$ au produit $(p-1)(q-1)$. (Vous vous étiez posé la question, n'est-ce pas ?) Puisque $p$ est premier, si $t \mod p$ est différent de zéro, alors $(t \mod p)^{p-1} \mod p = 1$.

Rappelons que nous avons défini $e$ et $d$ afin qu'ils soient des inverses multiplicatifs, modulo $r$ : $ed \mod r = 1$. Autrement dit, $ed = 1 + h(p-1)(q-1)$ pour un entier $h$. Si $t \mod p$ est différent de zéro, alors nous pouvons écrire :

$$\begin{aligned} t^{ed} \mod p &= (t \mod p)^{ed} \mod p \\ &= (t \mod p)^{1+h(p-1)(q-1)} \mod p \\ &= \left((t \mod p) \cdot \left((t \mod p)^{p-1} \mod p\right)^{h(q-1)}\right) \mod p \\ &= (t \mod p) \cdot (1^{h(q-1)} \mod p) \\ &= t \mod p \end{aligned}$$

Bien entendu, si $t \mod p$ est égal à 0, alors $t^{ed} \mod p$ est égal à 0.

Un raisonnement comparable montre que si $t \mod q$ est différent de 0, alors $t^{ed} \mod q$ est égal à $t \mod q$ et que si $t \mod q$ est égal à 0, alors $t^{ed} \mod q$ est égal à 0.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Pour aller au bout de notre démonstration, nous devons nous servir d'un autre résultat de la théorie des nombres : puisque $p$ et $q$ sont premiers entre eux (chacun étant premier), si $x \bmod p = y \bmod p$ et si $x \bmod q = y \bmod q$, alors $x \bmod pq = y \bmod pq$. (Ce résultat vient du théorème des restes chinois.) En remplaçant $x$ par $t^{ed}$ et $y$ par $t$, en sachant que $n = pq$ et que $t$ est inférieur à $n$, alors nous avons $t^{ed} \bmod n = t \bmod n = t$. C.Q.F.D. !

## Cryptosystèmes hybrides

Même si l'arithmétique sur de grands nombres est envisageable, la rapidité d'exécution va en souffrir. Le chiffrement et le déchiffrement d'un long message, qui contient des centaines ou des milliers de blocs de texte en clair, peuvent demander un temps assez long. RSA est souvent employé dans un système hybride, en partie à clé publique et en partie à clé symétrique.

Voici comment vous pouvez m'envoyer un message chiffré avec un système hybride. Nous nous mettons d'accord sur le système à clé publique et sur le système à clé symétrique que nous employons ; disons RSA et AES. Vous choisissez une clé $k$ pour AES et vous la chiffrez avec ma clé publique RSA. Vous obtenez $F_P(k)$. En utilisant la clé $k$, vous soumettez ensuite les blocs de texte en clair à AES afin d'obtenir des blocs de texte chiffré. Vous m'envoyez $F_P(k)$ et les blocs de texte chiffré. Je déchiffre $F_P(k)$ en calculant $F_S(F_P(k))$, ce qui me permet de récupérer la clé $k$. J'utilise ensuite cette clé pour déchiffrer les blocs de texte chiffré avec AES, obtenant ainsi les blocs de texte en clair. Si nous avions décidé d'utiliser le chaînage de blocs et si nous avions eu besoin d'un vecteur d'initialisation, alors vous auriez pu le chiffrer avec RSA ou AES.

## Générer des nombres aléatoires

Nous l'avons vu, certains cryptosystèmes ont besoin de générer des nombres aléatoires – plus précisément, des entiers aléatoires strictement positifs. Puisque nous représentons un entier par une suite de bits, nous devons en réalité être capable de générer des bits aléatoires, que nous interprétons ensuite comme un entier.

Les bits aléatoires peuvent uniquement provenir d'un processus aléatoire. Comment pouvons-nous faire en sorte qu'un programme qui s'exécute sur un ordinateur soit un processus aléatoire ? Très souvent, ce n'est pas possible, car un programme informatique est constitué d'instructions déterministes parfaitement définies, qui produisent toujours le même résultat à partir des mêmes données. Pour aider les logiciels de cryptographie,

Copyright © 2013 Dunod.

certains processeurs modernes disposent d'une instruction qui génère des bits aléatoires à partir d'une source aléatoire, comme un bruit thermique dans les circuits. Les concepteurs de ces processeurs font face à un défi triple : générer les bits suffisamment rapidement pour les applications qui ont besoin de nombres aléatoires, garantir que les bits générés satisfont aux tests statistiques de base qui caractérisent leur nature aléatoire, et demander une quantité de puissance raisonnable pour la génération et le test des bits aléatoires.

Les logiciels de cryptographie obtiennent généralement les bits à partir d'un **générateur de nombres pseudo-aléatoires** (PRNG, *PseudoRandom Number Generator*). Un **PRNG** est un programme déterministe qui produit une séquence de valeurs à partir d'une valeur initiale, ou **graine**, et d'une règle déterministe programmée qui stipule comment produire la valeur suivante de la séquence à partir de la valeur courante. Si un PRNG est démarré à chaque fois avec la même graine, la même séquence de valeurs est produite. Ce comportement reproductible est intéressant pour le débogage, mais plutôt gênant pour la cryptographie. Les standards récents pour les générateurs de nombres aléatoires dans les cryptosystèmes imposent des implémentations spécifiques des PRNG.

Si nous utilisons un PRNG pour générer des bits qui semblent aléatoires, nous devons le démarrer à chaque fois avec une graine différente, et celle-ci doit être aléatoire. En particulier, elle doit se fonder sur des bits impartiaux (sans favoritisme envers le 0 ou le 1), indépendants (quelles que soient les informations dont nous disposons sur les bits précédemment générés, nous avons une chance sur deux de deviner le bit suivant) et non prévisibles pour l'ennemi qui tente de casser notre cryptosystème. Si le processeur de notre machine dispose d'une instruction qui génère des bits aléatoires, elle conviendra à la création de la graine du PRNG.

## Lectures complémentaires

La cryptographie est l'une des composantes de la sécurité des systèmes informatiques. L'ouvrage de Smith et Marchesini [SM08] aborde ce sujet de long en large, ainsi que la manière d'attaquer les cryptosystèmes. Pour aller plus loin dans la cryptographie, nous vous conseillons les livres de Katz et Lindell [KL08] et de Menezes, van Oorschot et Vanstone [MvOV96]. Le Chapitre 31 du CLRS [CLRS09] pose les bases de la théorie des nombres utilisées en cryptographie et présente RSA ainsi que le test de primalité de Miller-Rabin. Diffie et Hellman [DH76] ont proposé la cryptographie à clé publique en 1976. L'article original de Rivest, Shamir et Adelman [RSA78], qui décrit RSA, a été publié deux ans plus tard.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Pour de plus amples informations sur les PRNG, consultez l'Annexe C du Federal Information Processing Standards Publication 140-2 [FIP11]. L'article de Taylor et Cox [TC11] explique comment réaliser une implémentation matérielle d'un générateur de nombres aléatoires fondé sur le bruit thermique.

Copyright © 2013 Dunod.

# 9 Compression de données

Au chapitre précédent, nous avons vu comment transformer des informations afin de les protéger contre les mauvaises intentions d'un ennemi. Mais il existe d'autres raisons de vouloir transformer des données. L'objectif sera parfois de les améliorer, par exemple en retouchant une image à l'aide d'un logiciel comme Adobe Photoshop pour retirer les yeux rouges ou modifier les teintes de la peau. Nous pourrions également vouloir ajouter une redondance des informations afin que les erreurs puissent être détectées et corrigées en cas de bits incorrects.

Dans ce chapitre nous étudions une autre façon de transformer des informations : en les compressant. Avant de présenter les méthodes utilisées pour compresser et décompresser des informations, nous devons répondre à trois questions :

1. Pourquoi voudrions-nous compresser des informations ?

   Il existe en général deux raisons de compresser des informations : gagner du temps et/ou de l'espace.

   Lorsque les données sont transmises au travers d'un réseau, plus le nombre de bits est faible, plus la transmission est rapide. L'expéditeur compresse donc souvent les données avant de les envoyer et le destinataire décompresse celles qu'il reçoit.

   Lorsque l'espace de stockage disponible peut limiter la quantité d'informations à enregistrer, leur compression permettra d'en enregistrer un plus grand nombre. Par exemple, les formats MP3 et JPEG compressent les sons et les images, mais de façon que le quidam ne voit aucune différence, si tant est qu'il en existe une, entre la version d'origine et la version compressée.

2. Quelle est la qualité des informations compressées ?

   Les méthodes de compression peuvent être avec ou sans perte. Dans le cas d'une **compression sans perte**, la décompression des informations compressées permet de retrouver les données d'origine. Dans

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

le cas d'une **compression avec perte**, les informations décompressées diffèrent des données d'origine, mais, dans l'idéal, de manière non significative. La compression MP3 et JPEG conduit à des pertes d'informations, contrairement à la méthode mise en œuvre par le programme zip.

De façon générale, la compression du texte doit se faire sans perte. En effet, dans ce cas, une différence d'un seul bit peut être significative. Les phrases suivantes diffèrent d'un seul bit au niveau des codes ASCII des lettres[1] :

```
On ne sait jamais l'étendue du tort que l'on fait.
On ne sait jamais l'étendue du port que l'on fait.
```

3. Pourquoi est-il possible de compresser les informations ?

Pour la compression avec perte, la réponse est facile : nous tolérons simplement une diminution de la précision. Qu'en est-il de la compression sans perte ? Les informations numériques contiennent souvent des bits redondants ou inutiles. Par exemple, chaque caractère ASCII occupe un octet de 8 bits et les caractères les plus utilisés (hormis les lettres accentuées) ont tous 0 pour bit de poids fort (celui à l'extrême gauche). Autrement dit, les codes des caractères ASCII vont de 0 à 255, mais les caractères les plus utilisés se trouvent tous dans la plage 0 à 127. Dans un texte ASCII, le huitième bit est donc souvent inutile et il est facile de le compresser afin d'en réduire la taille de 12,5 %.

Prenons un exemple représentatif de l'utilisation de la redondance dans la compression sans perte : la transmission d'un document en noir et blanc, comme le font les télécopieurs. Ces machines transmettent un document sous forme d'une suite de points noirs ou blancs qui forment le contenu. La plupart des télécopieurs procèdent de haut en bas, ligne par ligne. Lorsque le document est essentiellement constitué de texte, la plus grande partie de la feuille est blanche et chaque ligne contiendra probablement de nombreux éléments blancs consécutifs. Si une ligne contient une partie d'un trait horizontal noir, elle peut avoir plusieurs éléments noirs consécutifs. Au lieu d'indiquer individuellement la couleur de chaque élément d'une séquence, les télécopieurs compressent les informations en indiquant la longueur de la séquence d'éléments et leur couleur. Par exemple, une suite de 140 éléments blancs est compressée sous la forme des onze bits suivants : 10010001000.

La compression des données est un domaine qui fait l'objet d'études approfondies et nous ne pourrons donc que l'aborder dans cet ouvrage.

1. Les codes ASCII de `p` et de `t` sont, respectivement, 01110000 et 01110100.

Copyright © 2013 Dunod.

Nous allons nous focaliser sur la compression sans perte, mais la section « Lectures complémentaires » propose deux bons ouvrages sur la compression avec perte.

Contrairement aux chapitres précédents, nous n'allons pas nous intéresser de près aux temps d'exécution, même si nous les mentionnerons au moment opportun. Nous sommes beaucoup plus intéressés par la taille des informations compressées que par le temps qu'il faut pour les obtenir.

## Codages de Huffman

Revenons aux chaînes de caractères qui représentent l'ADN. Au Chapitre 7, nous avons expliqué que les biologistes représentent l'ADN sous forme de chaînes constituées à partir de quatre caractères : `A`, `C`, `G` et `T`. Supposons que nous ayons un brin d'ADN représenté par $n$ caractères, dont 45 % sont des `A`, 5 % des `C`, 5 % des `G` et 45 % des `T`, avec un ordre d'apparition quelconque. Si le brin est représenté à l'aide du jeu de caractères ASCII, chaque caractère occupe huit bits et il faut donc $8n$ bits pour représenter l'intégralité du brin. Nous pouvons évidemment faire plus court. Puisque nous représentons les brins d'ADN à l'aide de seulement quatre caractères, nous n'avons besoin que de deux bits pour les coder (00, 01, 10, 11) et pouvons ainsi réduire l'espace requis à $2n$ bits.

Mais nous pouvons faire encore mieux en exploitant les fréquences relatives des caractères. Utilisons les séquences de bits suivantes pour coder les caractères : `A` = 0, `C` = 100, `G` = 101, `T` = 11. Les caractères les plus fréquents ont les suites de bits les plus courtes. Le brin `TAATTAGAAATTCTATTATA` de 20 caractères peut être encodé par la séquence 110 01111010100011111001101111010 de 33 bits. (Nous expliquerons plus loin pourquoi nous avons choisi cet encodage précis et donnerons ses propriétés.) En raison des fréquences d'apparition des quatre caractères, pour encoder le brin de $n$ caractères, nous avons uniquement besoin de $0{,}45 \cdot n \cdot 1 + 0{,}05 \cdot n \cdot 3 + 0{,}05 \cdot n \cdot 3, +0{,}45 \cdot n \cdot 2 = 1{,}65n$ bits. (Pour l'exemple précédent, $33 = 1{,}65 \cdot 20$.) En profitant des fréquences relatives des caractères, nous faisons donc mieux que $2n$ bits !

Avec le codage choisi, les caractères les plus utilisés ont des suites de bits plus courtes. Il présente également une caractéristique intéressante : aucun code n'est le préfixe d'un autre. Le code de `A` est 0, et aucun autre code ne commence par 0. Le code de `T` est 11, et aucun autre code ne commence par 11. Etc. Un tel codage est appelé **codage préfixe**[1] (*prefix-free*).

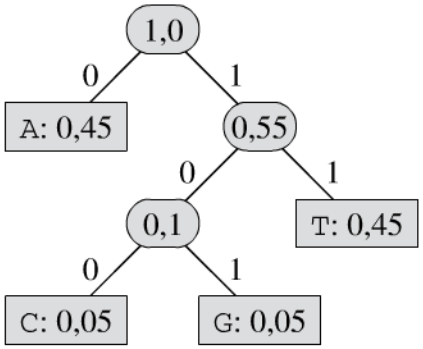
1. Sans doute « codage sans préfixe » serait-il plus adapté, mais le terme « codage préfixe » est standard dans la littérature.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Le principal avantage des codages préfixes apparaît lors de la décompression. Puisqu'aucun code n'est le préfixe d'un autre, nous pouvons établir sans ambiguïté une correspondance entre les bits compressés et le caractère d'origine pendant la décompression des bits dans l'ordre d'apparition. Prenons par exemple la séquence compressée 110011110101000111 110011011110110. Aucun caractère n'est codé avec un seul bit à 1 et seul le code de `T` commence par 11. Nous savons donc que le premier caractère du texte décompressé doit être `T`. En retirant 11, nous obtenons la séquence 001111010100011111001101111 0110. Seul le code de `A` commence par 0 et le premier caractère de cette séquence doit donc être `A`. Nous retirons le 0 et les bits 011110 suivants qui correspondent à `ATTA`. La séquence devient 101000111110011011110110. Puisque seul le code de `G` commence par 101, le caractère décompressé suivant est `G`. Etc.

Si nous mesurons l'efficacité des méthodes de compression en fonction de la longueur moyenne des informations compressées, alors, parmi les codages préfixes, ceux de Huffman[1] sont les meilleurs. Ils ont toutefois un inconvénient : la fréquence de tous les caractères est supposée connue à l'avance. La compression se fait donc souvent en deux passes sur le texte non compressé : la première pour déterminer la fréquence des caractères, la seconde pour associer un code à chaque caractère. Nous verrons plus loin comment éviter la première passe, au prix de calculs supplémentaires.

Lorsque la fréquence des caractères est connue, la méthode de Huffman construit un arbre binaire (voir page 103) qui montre comment former le codage. Il se révélera également pratique pour la décompression. Voici l'arbre qui correspond à notre exemple d'encodage de l'ADN :




Les feuilles de l'arbre, tracées sous forme de rectangles, représentent les caractères, avec leur fréquence indiquée. Les **nœuds internes** sont dessinés avec des angles arrondis. Ils comprennent la somme des fréquences des feuilles qui se trouvent en dessous. Nous verrons plus loin pourquoi il est intéressant de mémoriser les fréquences dans les nœuds internes.

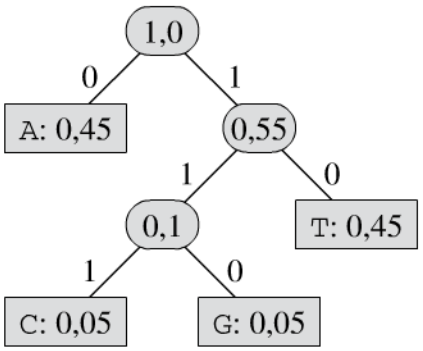
À côté de chaque arête de l'arbre apparaît un 0 ou un 1. Pour déterminer le code d'un caractère, il suffit de suivre le chemin à partir de la

1. Nommés d'après le nom de leur inventeur, David Huffman.

Copyright © 2013 Dunod.

racine jusqu'à la feuille du caractère et de concaténer les bits rencontrés au fur et à mesure. Par exemple, pour déterminer le code de G, nous partons de la racine et prenons la branche libellée 1 vers l'enfant de droite, puis la branche libellée 0 vers l'enfant de gauche (le nœud interne de fréquence 0,1), et enfin la branche libellée 1 vers le nœud de droite (la feuille qui correspond à G). La concaténation de ces bits donne le code 101 pour G.

Nous avons libellé 0 l'arête qui va vers l'enfant gauche et 1 celle qui va vers l'enfant droit, mais les libellés n'ont pas vraiment d'importance. Nous aurions pu avoir l'arbre suivant :



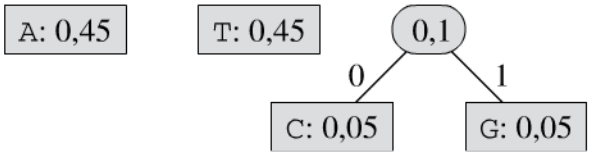

Dans ce cas, les codes auraient été A = 0, C = 111, G = 110 et T = 10. Il s'agit toujours d'un codage préfixe et le nombre de bits de chaque code reste identique au précédent. En effet, le nombre de bits du code d'un caractère correspond à la **profondeur** de la feuille du caractère : le nombre d'arêtes qui composent le chemin allant de la racine à la feuille. Le travail est toutefois plus simple si les arêtes qui vont aux enfants de gauche sont libellées 0 et si celles qui vont aux enfants de droite sont libellées 1.

Lorsque nous connaissons les fréquences des caractères, nous construisons l'arbre binaire de bas en haut. Nous partons de chacun des $n$ nœuds feuilles, qui correspond au caractère non compressé, comme s'il s'agissait d'un arbre individuel. Initialement, chaque feuille est donc également une racine. Nous recherchons ensuite de manière répétée les deux nœuds racines qui ont les fréquences les plus faibles, nous créons une nouvelle racine avec ces deux nœuds comme enfants et lui affectons la somme des fréquences de ces enfants. La procédure se poursuit jusqu'à ce que toutes les feuilles se trouvent sous une seule racine. Au fur et à mesure, nous libellons chaque arête vers le nœud gauche avec 0 et chaque arête vers le nœud droit avec 1, même si, après que nous avons sélectionné les deux racines qui ont les fréquences les plus faibles, peu importe celle qui devient l'enfant de gauche et celle qui devient l'enfant de droite de la nouvelle racine.

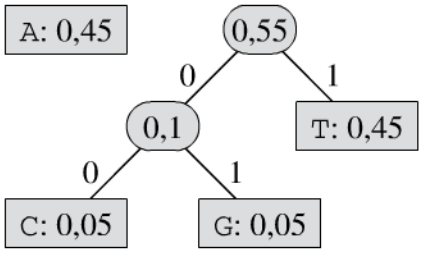
Avec notre exemple de brin d'ADN, voici le déroulement de la procédure. Nous partons avec quatre nœuds, chacun étant une feuille qui représente un caractère :

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

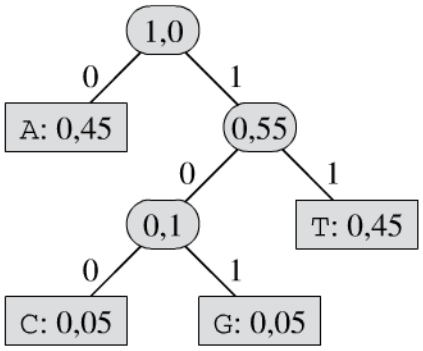
Les nœuds de C et de G ont les fréquences les plus faibles et nous créons donc un nouveau nœud dont les enfants sont ces deux nœuds. Nous lui donnons la somme de leur fréquence :




Parmi les trois racines restantes, celle que nous venons de créer a la fréquence la plus faible, 0,1, et les deux autres ont une fréquence égale à 0,45. Peu importe le nœud que nous choisissons comme seconde racine. Nous optons pour celui de T et nous en faisons, avec la racine de fréquence 0,1, les enfants d'un nouveau nœud dont la fréquence 0,55 correspond à leur somme :




Il ne reste plus que deux racines. Nous créons un nouveau nœud avec ces racines en tant qu'enfants et avec la fréquence 1,0 (bien que nous n'en aurons pas besoin) :




Puisque toutes les feuilles sont à présent sous cette nouvelle racine, la construction de l'arbre binaire est terminée.

Soyons plus précis en définissant une procédure qui construit l'arbre binaire. CONSTRUIRE-ARBRE-HUFFMAN prend en entrée deux tableaux de $n$ éléments, *car* et *fréq*, où *car*[$i$] contient le $i$-ème caractère non compressé et *fréq*[$i$] donne la fréquence de ce caractère. Elle prend également en paramètre la valeur de $n$. Pour rechercher les deux racines qui possèdent les fréquences les plus faibles, elle appelle les procédures INSÉRER et EXTRAIRE-MIN d'une file de priorité (voir page 102).

Copyright © 2013 Dunod.

*Procédure* CONSTRUIRE-ARBRE-HUFFMAN(*car*, *fréq*, *n*)
*Entrées* :

- *car :* un tableau de $n$ caractères non compressés.
- *fréq :* un tableau de $n$ fréquences de caractères.
- $n$ : la taille des tableaux *car* et *fréq*.

*Sortie* : La racine de l'arbre binaire construit pour les codages de Huffman.

1. Soit $Q$ une file de priorité vide.
2. Pour $i = 1$ à $n$ :
   A. Construire un nouveau nœud $z$ qui contient *car*[$i$] et dont la fréquence est *fréq*[$i$].
   B. Appeler INSÉRER($Q$, $z$).
3. Pour $i = 1$ à $n - 1$ :
   A. Appeler EXTRAIRE-MIN($Q$) et affecter le nœud extrait à $x$.
   B. Appeler EXTRAIRE-MIN($Q$) et affecter le nœud extrait à $y$.
   C. Construire un nouveau nœud $z$ dont la fréquence est la somme des fréquences de $x$ et de $y$.
   D. Fixer l'enfant de gauche de $z$ à $x$ et celui de droite à $y$.
   E. Appeler INSÉRER($Q$, $z$).
4. Appeler EXTRAIRE-MIN($Q$) et retourner le nœud extrait.

Lorsque la procédure arrive à l'étape 4, il ne reste plus qu'un seul nœud dans la file de priorité. Il s'agit de la racine de l'arbre binaire complet.

Vous pouvez dérouler cette procédure avec les arbres binaires précédents. Les racines dans la file de priorité au début de chaque itération de la boucle de l'étape 3 apparaissent en haut de chaque figure.

Analysons rapidement le temps d'exécution de CONSTRUIRE-ARBRE-HUFFMAN. Si l'on suppose que la file de priorité est implémentée à l'aide d'un tas binaire, chaque opération INSÉRER et EXTRAIRE-MIN prend un temps $O(\lg n)$. La procédure appelle chacune de ces opérations à $2n - 1$ reprises, pour un temps total $O(n \lg n)$. Les autres opérations demandent un temps total $\Theta(n)$ et la procédure CONSTRUIRE-ARBRE-HUFFMAN s'exécute donc en un temps $O(n \lg n)$.

J'ai mentionné précédemment que la décompression sera plus simple si nous disposons de l'arbre binaire créé par CONSTRUIRE-ARBRE-HUFFMAN. En partant de la racine de l'arbre binaire, nous allons vers les feuilles en fonction des bits des informations compressées. Nous extrayons chaque bit et allons vers la gauche s'il s'agit d'un 0, et vers la droite s'il s'agit d'un 1. Lorsque nous arrivons à une feuille, nous nous arrêtons, produisons le caractère correspondant et reprenons la recherche à la racine. Revenons à

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

notre exemple de l'ADN et décompressons la suite de bits 1100111101010 0011111001101111011. Nous retirons le premier 1 et partons à droite de la racine. Nous retirons un autre 1 et allons de nouveau à droite, pour arriver à la feuille du caractère `T`. Nous produisons un `T` et reprenons la recherche à la racine. Nous retirons le bit suivant, 0, et allons vers la gauche, pour arriver à la feuille du caractère `A`, que nous produisons. Nous retournons à la racine. La décompression se poursuit de cette manière, jusqu'à ce que tous les bits de l'information compressée aient été traités.

Si l'arbre binaire est déjà créé avant la décompression, le traitement de chaque bit se fait en un temps constant. Comment la procédure de décompression peut-elle avoir accès à cet arbre binaire ? Une possibilité est d'en ajouter une représentation à l'information compressée. Une autre consiste à fournir une table de décodage avec les informations traitées. Chaque entrée de la table comprend le caractère, le nombre de bits de son code et le code lui-même. À partir de cette table, il est possible de générer l'arbre binaire en un temps linéaire qui dépend du nombre total de bits dans tous les codes.

La procédure CONSTRUIRE-ARBRE-HUFFMAN est un exemple d'**algorithme glouton** dans lequel nous prenons la décision qui semble la meilleure à un moment donné. Puisque nous voulons que les caractères les moins fréquents soient les plus éloignés de la racine de l'arbre binaire, l'approche gloutonne sélectionne toujours les deux racines de plus faible fréquence afin de les placer sous un nouveau nœud, qui peut ensuite devenir un enfant d'un autre nœud. L'algorithme de Dijkstra (voir page 97) est un autre algorithme glouton, car il relâche toujours les arcs qui partent du sommet de plus courte valeur parmi ceux encore présents dans la file de priorité.

J'ai implémenté le codage de Huffman et l'ai exécuté sur une version en ligne de *Moby Dick*. Le texte d'origine comprend 1 193 826 octets. Sa version compressée en occupe uniquement 673 579, c'est-à-dire 56,42 % de la taille initiale, sans l'encodage lui-même. Nous pouvons l'exprimer d'une autre manière : en moyenne, le codage de chaque caractère nécessite uniquement 4,51 bits. Sans trop de surprise, le caractère le plus fréquent était l'espace (15,96 %), suivi du `e` (9,56 %). Les moins fréquents, apparus uniquement à deux reprises chacun, étaient `$`, `&`, `[` et `]`.

## Codages de Huffman adaptatifs

Dans la pratique, nombreux sont ceux à trouver que les deux passes sur l'entrée, une pour calculer les fréquences et une pour encoder les caractères, prennent trop de temps. C'est pourquoi, les programmes de compression et de décompression travaillent de façon adaptative, en actualisant les fréquences de caractères et l'arbre binaire au fur et à mesure de la compression ou de la décompression, tout cela au cours d'une seule et même passe.

Copyright © 2013 Dunod.

Le programme de compression débute avec un arbre binaire vide. Chaque caractère lu depuis l'entrée peut, soit déjà se trouver dans l'arbre binaire, soit être nouveau. Dans le premier cas, le programme de compression produit le code du caractère en fonction de l'arbre actuel, incrémente la fréquence du caractère et, si nécessaire, met à jour l'arbre afin de tenir compte de la nouvelle fréquence. Si le caractère est absent de l'arbre binaire, le programme de compression produit le caractère non codé (tel quel), l'ajoute à l'arbre binaire et actualise celui-ci en conséquence.

Le programme de décompression opère de manière comparable. Il construit également un arbre binaire pendant le traitement des informations compressées. Lorsqu'il rencontre les bits d'un caractère dans l'arbre binaire, il traverse celui-ci afin de déterminer le caractère correspondant, produit ce caractère, augmente sa fréquence et actualise l'arbre. Lorsqu'il rencontre un caractère absent de l'arbre, il produit ce caractère, l'ajoute à l'arbre et actualise celui-ci.

Il manque toutefois quelque chose dans ces explications. Les bits ne sont que des bits, qu'ils représentent des caractères ASCII ou un codage de Huffman. Comment le programme de décompression peut-il déterminer si les bits qu'il examine représentent un caractère codé ou non codé ? La séquence 101 représente-t-elle le caractère actuellement encodé 101 ou s'agit-il du début d'un caractère non encodé sur 8 bits ? La solution à ce problème consiste à placer un **code d'échappement** avant chaque caractère non encodé. Il s'agit d'un code spécial qui indique que la séquence de bits suivante représente un caractère non encodé. Si le texte d'origine comprend $k$ caractères différents, alors seulement $k$ codes d'échappement apparaîtront dans l'information compressée, chacun avant la première occurrence d'un caractère. Puisque les codes d'échappement seront généralement peu fréquents, nous n'avons pas besoin de leur affecter des séquences de bits courtes au détriment d'autres caractères plus employés. Pour garantir que les codes d'échappement ne sont pas courts, une bonne solution consiste à inclure un caractère de code d'échappement dans l'arbre binaire en gardant sa fréquence à 0. L'actualisation de l'arbre binaire changera la séquence de bits du code d'échappement dans les programmes de compression et de décompression, mais la feuille associée sera toujours la plus éloignée de la racine.

## TÉLÉCOPIEURS

Nous avons mentionné précédemment que les télécopieurs compressent l'information en indiquant les couleurs et les longueurs des suites d'éléments identiques qui composent les lignes du document transmis. Ce principe se nomme **codage par plage** (RLE, *Run-Length Encoding*). Ces machines utilisent à la fois le codage par plage et le codage de Huffman. Dans la norme qui définit le fonctionnement des télécopieurs sur ligne

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

téléphonique normale, cent quatre codes indiquent des séquences de différentes longueurs d'éléments blancs et cent quatre codes indiquent des séquences de différentes longueurs d'éléments noirs. Les séquences d'éléments, qu'ils soient noirs ou blancs, utilisent des codages préfixes, même si des codes de séquences d'éléments blancs sont des préfixes de codes de séquences d'éléments noirs, et *vice versa*.

Pour déterminer les codes des séquences, un comité de normalisation est parti de huit documents représentatifs et a compté la fréquence de chaque séquence. Il a ensuite créé les codages de Huffman correspondants. Les séquences les plus fréquentes, et donc les codes les plus courts, sont constituées de deux, trois et quatre éléments noirs, avec les codes respectifs 11, 10 et 011. Les autres séquences répandues sont un élément noir (010), cinq et six éléments noirs (0011 et 0010), deux à sept éléments blancs (tous avec des codes de quatre bits), ainsi que d'autres séquences relativement courtes. L'une séquence plutôt fréquente comprend 1 664 éléments blancs, qui représente une ligne intégralement composée d'éléments blancs. D'autres codes courts correspondent à des séquences d'éléments blancs dont les longueurs sont des puissances de deux ou des sommes de deux puissances de deux (comme 192, qui est égal à $2^7 + 2^6$). Des séquences peuvent être encodées par concaténation des codes de séquences plus courtes. Nous avons précédemment donné un exemple de code pour une séquence de 140 éléments blancs, 10010001000. Il s'agit en réalité de la concaténation du code d'une séquence de 128 éléments blancs (10010) et de celui d'une séquence de 12 éléments blancs (001000).

Outre la compression des informations en opérant à partir des lignes du document, certains télécopieurs procèdent à la compression dans les deux dimensions. Des séquences d'éléments de même couleur peuvent apparaître aussi bien verticalement qu'horizontalement. Au lieu de traiter chaque ligne comme si elle était isolée, l'encodage se fait sur les différences entre une ligne et la précédente. Dans la plupart des cas, ces différences se résument à quelques éléments. Mais cette approche comprend un risque de propagation des erreurs : une erreur d'encodage ou de transmission peut conduire à plusieurs lignes consécutives incorrectes. C'est pourquoi les télécopieurs qui se fondent sur ce principe et transmettent les documents au travers de lignes téléphoniques l'appliquent à un nombre limité de lignes consécutives. Après un certain nombre de lignes, ils transmettent l'intégralité d'une ligne en utilisant le codage de Huffman au lieu de transmettre uniquement les différences avec la ligne précédente.

## Compression LZW

Une autre méthode de compression sans perte, notamment employée avec le texte, tire profit de la récurrence des informations dans le contenu,

Copyright © 2013 Dunod.

même si elles se trouvent à des emplacements non consécutifs. Prenons par exemple une citation du discours inaugural de John F. Kennedy :

> *« Ne vous demandez pas ce que votre pays peut faire pour vous, mais demandez-vous ce que vous pouvez faire pour votre pays. »*

Si certains mots de cette citation ne sont présents qu'une seule fois, d'autres apparaissent à plusieurs reprises. Imaginons que nous construisions la table des mots suivante :

| Indice | Mot |
|---|---|
| 1 | ne |
| 2 | vous |
| 3 | demandez |
| 4 | pas |
| 5 | ce |
| 6 | que |
| 7 | votre |
| 8 | pays |
| 9 | peut |
| 10 | faire |
| 11 | pour |
| 12 | mais |
| 13 | pouvez |

Nous pouvons alors encoder cette citation de la manière suivante (en ignorant les majuscules et la ponctuation) :

1 2 3 4 5 6 7 8 9 10 11 2 12 3 2 5 6 2 13 10 11 7 8

Puisque cette citation ne comprend que quelques mots et qu'un octet peut contenir des entiers dans la plage 0 à 255, nous pouvons stocker chaque indice dans un seul octet. Nous pouvons donc placer cette citation dans uniquement 23 octets, un pour chaque mot, plus l'espace nécessaire à la table. Avec un caractère par octet, la citation d'origine, sans la ponctuation mais avec les espaces entre les mots, nécessite 120 octets.

Bien entendu l'espace occupé par la table a de l'importance, sinon nous pourrions simplement numéroter chaque mot possible et compresser un fichier en stockant uniquement les indices des mots. Pour certains mots, cette approche augmente la taille, tandis que pour d'autres elle la réduit. Pourquoi ? Supposons qu'il existe moins de $2^{32}$ mots afin que nous puissions mémoriser chaque indice sur 32 bits. Nous pouvons représenter chaque mot par 4 octets et cette approche est donc perdante pour les mots de trois lettres ou moins, car ils nécessitent un octet de moins par lettre non compressée.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Le véritable obstacle à la numérotation des mots possibles réside toutefois dans le fait qu'un texte peut employer des « mots » qui ne sont pas des mots ou, plus précisément, pas des mots français. C'est par exemple le cas du premier quatrain du poème *Jabberwocky* de Lewis Carroll :

> Il était grilheure ; les slictueux toves
> Sur l'alloinde gyraient et vriblaient ;
> Tout flivoreux étaient les borogoves
> Les vergons fourgus bourniflaient.

Les programmes informatiques, dans lesquels les noms des variables ne sont pas toujours des mots français, présentent cette même caractéristique. Ajoutons à cela les majuscules, la ponctuation et les noms de lieux très longs[1], et nous constatons que la compression du texte par numérotation de chaque mot possible va demander un très grand nombre d'indices. Certainement plus que $2^{32}$ et, puisque n'importe quelle combinaison de caractères peut apparaître dans un texte, en réalité une infinité.

Cependant, tout n'est pas perdu. Il est tout à fait possible d'exploiter la récurrence des informations. Il suffit simplement de ne pas faire une telle fixation sur les mots récurrents. N'importe quelle séquence identique de caractères conduira à une amélioration. Plusieurs méthodes de compression se fondent sur la récurrence des séquences de caractères. Celle que nous allons examiner se nomme **LZW**[2] ; on la retrouve dans de nombreux programmes de compression employés quotidiennement.

LZW effectue une seule passe sur l'entrée, que ce soit pour la compression ou la décompression. Dans les deux cas, l'algorithme construit un dictionnaire des séquences de caractères qu'il rencontre et utilise des indices dans ce dictionnaire pour représenter les séquences. Le dictionnaire peut être vu comme un tableau de chaînes de caractères. Puisque nous pouvons définir des indices dans ce tableau, nous pouvons parler de sa $i$-ème entrée. Au début des informations fournies en entrée, les séquences ont tendance à être courtes et leur représentation par des indices risque d'augmenter la taille au lieu de la réduire. Mais, au fur et à mesure du traitement des caractères d'entrée, les séquences du dictionnaire deviennent de plus en plus longues et leur représentation par un indice permet de gagner un espace significatif. Par exemple, j'ai soumis la version anglaise du texte de *Moby Dick* à un programme de compression LZW. Dans la sortie obtenue, un indice représentant la séquence de dix caractères « ␣from ␣the ␣ » est apparu à

1. Comme celui d'un village gallois : Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch.
2. Vous l'avez probablement deviné, ainsi nommée en raison du nom de ses inventeurs. Terry Welch a créé LZW en modifiant la méthode de compression LZ78 qu'avaient proposée Abraham Lempel et Jacob Ziv.

Copyright © 2013 Dunod.

20 reprises (␣ est une espace). Elle comprenait également trente-trois fois un indice représentant la séquence de huit caractères « ␣of␣the␣ ».

La compression et la décompression ajoutent au dictionnaire une séquence d'un caractère pour chaque caractère du jeu choisi. Si le jeu de caractères ASCII est utilisé, le dictionnaire commence avec 256 séquences d'un seul caractère ; la $i$-ème entrée du dictionnaire correspond au caractère dont le code ASCII est $i$.

Avant d'expliquer le fonctionnement général de la compression, examinons deux cas. La procédure de compression construit des chaînes de caractères, les insère dans le dictionnaire et envoie en sortie des indices dans le dictionnaire. Supposons qu'elle commence par construire une chaîne avec le caractère `T` lu sur l'entrée. Puisque le dictionnaire comprend chaque séquence d'un seul caractère, la procédure y trouve `T`. Chaque fois qu'elle trouve dans le dictionnaire la chaîne de caractères qu'elle construit, la procédure prend le caractère suivant sur l'entrée et l'ajoute à la chaîne en cours de construction. Supposons que ce prochain caractère soit `A`. La procédure de compression ajoute donc `A` à la chaîne qu'elle construit, qui devient `TA`. Supposons que `TA` soit également dans le dictionnaire. La procédure lit ensuite le caractère sur l'entrée, disons `G`. Elle ajoute `G` à la chaîne en cours de construction, qui devient `TAG`. Supposons cette fois-ci que `TAG` ne soit pas dans le dictionnaire. La procédure réalise alors trois choses : (1) elle envoie sur la sortie l'indice de la chaîne `TA`, (2) elle insère la chaîne `TAG` dans le dictionnaire et (3) elle démarre la construction d'une nouvelle chaîne, qui contient initialement uniquement le caractère (`G`) à l'origine de l'absence de la chaîne `TAG` dans le dictionnaire.

Voici à présent le fonctionnement général de la compression. La procédure produit une séquence d'indices dans le dictionnaire. La concaténation des chaînes à ces indices donne le texte d'origine. La procédure construit les chaînes du dictionnaire un caractère à la fois. Par conséquent, lorsqu'elle insère une chaîne dans le dictionnaire, celle-ci est identique à une chaîne déjà présente dans le dictionnaire, mais avec un caractère supplémentaire. La procédure gère une chaîne $s$ de caractères consécutifs de l'entrée, en maintenant l'invariant que le dictionnaire comprend toujours $s$ dans une entrée. Même si $s$ est constituée d'un seul caractère, elle apparaît dans le dictionnaire, car celui-ci est alimenté par une séquence d'un seul caractère pour chaque caractère du jeu employé.

Initialement, $s$ est uniquement le premier caractère de l'entrée. Après la lecture d'un nouveau caractère $c$, la procédure de compression vérifie si la chaîne $s\ c$, constituée par concaténation de $c$ à la fin de $s$, se trouve actuellement dans le dictionnaire. Dans l'affirmative, elle ajoute $c$ à la fin de $s$ et appelle le résultat $s$ ; autrement dit, elle affecte $s\ c$ à $s$. La procédure construit une chaîne plus longue qui finira par être insérée dans le dictionnaire. Dans la négative, $s$ existe dans le dictionnaire, mais pas $s\ c$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans ce cas, la procédure envoie sur la sortie l'indice de *s* dans le dictionnaire, insère *s c* dans la première entrée vacante du dictionnaire, et fixe *s* au caractère d'entrée *c*. En ajoutant *s c* au dictionnaire, la procédure de compression a ajouté une chaîne qui étend *s* d'un seul caractère et, en fixant *s* à *c*, elle reprend le processus de construction d'une chaîne à rechercher dans le dictionnaire. Puisque *c* est une chaîne d'un seul caractère dans le dictionnaire, la procédure maintient l'invariant qui stipule que *s* doit apparaître quelque part dans le dictionnaire. Lorsque l'entrée est vide, la procédure envoie sur la sortie l'indice de la dernière chaîne *s*.

Voici la procédure COMPRESSION-LZW.

*Procédure* COMPRESSION-LZW(*texte*)

*Entrée* : *texte :* une séquence de caractères provenant du jeu de caractères ASCII.

*Sortie* : Une séquence d'indices dans un dictionnaire.

1. Pour chaque caractère *c* du jeu de caractères ASCII :
   A. Insérer *c* dans le dictionnaire à l'indice qui correspond au code numérique ASCII de *c*.
2. Fixer *s* au premier caractère de *texte*.
3. Tant que *texte* n'est pas vide, faire la chose suivante :
   A. Prendre le caractère suivant de texte de l'affecter à *c*.
   B. Si *s c* se trouve dans le dictionnaire, alors fixer *s* à *s c*.
   C. Sinon (*s c* n'est pas encore dans le dictionnaire), faire la chose suivante :
      i. Envoyer sur la sortie l'indice de *s* dans le dictionnaire.
      ii. Insérer *s c* dans la prochaine entrée vacante du ii. dictionnaire.
      iii. Fixer *s* à la chaîne d'un seul caractère *c*.
4. Envoyer sur la sortie d'indices de *s* dans le dictionnaire.

Étudions-la sur la compression du texte `TATAGATCTTAATATA`. (La séquence `TAG` mentionnée précédemment sera rencontrée.) La table suivante montre ce qui se produit à chaque itération de la boucle de l'étape 3. Les valeurs de la chaîne *s* sont valables au début de l'itération.

| Itération | *s* | *c* | Sortie | Nouvelle chaîne dans le dictionnaire |
|---|---|---|---|---|
| 1 | T | A | 84 (T) | 256 : TA |
| 2 | A | T | 65 (A) | 257 : AT |
| 3 | T | A | | |

Copyright © 2013 Dunod.

| Itération | *s* | *c* | Sortie | Nouvelle chaîne dans le dictionnaire |
|---|---|---|---|---|
| 4 | TA | G | 256 (TA) | 258 : TAG |
| 5 | G | A | 71 (G) | 259 : GA |
| 6 | A | T | | |
| 7 | AT | C | 257 (AT) | 260 : ATC |
| 8 | C | T | 67 (C) | 261 : CT |
| 9 | T | T | 84 (T) | 262 : TT |
| 10 | T | A | | |
| 11 | TA | A | 256 (TA) | 263 : TAA |
| 12 | A | T | | |
| 13 | AT | A | 257 (AT) | 264 : ATA |
| 14 | A | T | | |
| 15 | AT | A | | |
| étape 4 | ATA | | 264 (ATA) | |

Après l'étape 1, le dictionnaire comprend dans les entrées 0 à 255 des chaînes d'un caractère pour chacun des 256 caractères ASCII. L'étape 2 affecte le premier caractère de l'entrée, T, à la chaîne *s*. Au cours de la première itération de la boucle principale de l'étape 3, *c* correspond au caractère suivant de l'entrée, c'est-à-dire A. La concaténation *s c* est donc la chaîne TA, qui ne se trouve pas encore dans le dictionnaire ; l'étape 3C s'exécute donc. Puisque la chaîne *s* contient uniquement T et que le code ASCII de T est 84, l'étape 3Ci envoie l'indice 84 sur la sortie. L'étape 3Cii insère la chaîne TA dans la prochaine entrée vacante du dictionnaire, qui se trouve à l'indice 256. L'étape 3Ciii reprend la construction de *s*, en lui affectant uniquement le caractère A. Au cours de la deuxième itération de la boucle de l'étape 3, *c* correspond au caractère suivant de l'entrée, T. La chaîne *s c* =AT n'est pas dans le dictionnaire et l'étape 3C envoie donc l'indice 65 (le code ASCII de A) sur la sortie, insère la chaîne AT dans l'entrée 257 et fixe *s* à T.

Les deux itérations suivantes de la boucle de l'étape 3 vont révéler l'intérêt du dictionnaire. Au cours de la troisième itération, *c* devient le caractère suivant de l'entrée, c'est-à-dire A. À présent, la chaîne *s c* =TA existe dans le dictionnaire et la procédure n'envoie rien sur la sortie. À la place, l'étape 3B ajoute le caractère d'entrée à la fin de *s*, qui devient TA. Lors de la quatrième itération, *c* est égal à G. La chaîne *s c* =TAG n'est pas dans le dictionnaire et l'étape 3Ci envoie sur la sortie l'indice 256 de *s*. Un nombre en sortie ne donne donc pas un mais deux caractères : TA.

Au terme de la procédure Compression-LZW, tous les indices du dictionnaire n'ont pas été envoyés sur la sortie, et certains peuvent avoir été

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

envoyés plusieurs fois. Si nous concaténons tous les caractères placés entre parenthèses dans la colonne de la sortie, nous obtenons le texte d'origine, TATAGATCTTAATATA.

Cet exemple est un petit peu trop court pour illustrer le réel bénéfice de la compression LZW. L'entrée occupe 16 octets, tandis que la sortie comprend 10 indices dans le dictionnaire. Chaque indice nécessite plus d'un octet. Si nous utilisons uniquement deux octets par indice dans la sortie, elle va occuper 20 octets. Si chaque indice est codé sur quatre octets, une taille fréquente pour les entiers, la sortie demande 40 octets.

Plus le texte d'entrée est long, meilleurs sont les résultats. La compression LZW permet de réduire la taille de *Moby Dick* de 1 193 826 octets à 919 012 octets. Dans ce cas, le dictionnaire comprend 230 007 entrées et les indices doivent être sur au moins quatre octets[1]. La sortie comprend 229 753 indices, ou 919 012 octets. Le taux de compression est inférieur à celui obtenu avec le codage de Huffman (673 579 octets), mais nous allons voir quelques solutions pour l'améliorer.

La compression LZW n'a d'intérêt que si la décompression des informations est possible. Heureusement, il est inutile d'enregistrer le dictionnaire avec les informations compressées. (Dans le cas contraire, le cumul de la sortie produite par la compression LZW et du dictionnaire conduirait non pas à une compression mais à une expansion, excepté si la récurrence des chaînes dans le texte d'origine est très élevée.) Comme nous l'avons mentionné précédemment, la décompression LZW reconstruit le dictionnaire à partir des informations compressées.

Voici comment la décompression LZW opère. À l'instar de la compression, la procédure remplit le dictionnaire avec les 256 séquences d'un seul caractère qui correspondent au jeu de caractères ASCII. Elle lit une suite d'indices donnés en entrée et construit le dictionnaire de façon comparable à la procédure de compression. Dès qu'elle produit une sortie, celle-ci provient d'une chaîne qui a été ajoutée au dictionnaire.

La plupart du temps, l'indice suivant lu correspond à une entrée qui se trouve déjà dans le dictionnaire (nous verrons plus loin ce qui se passe sinon). La procédure de décompression prend dans le dictionnaire la chaîne à cet indice et l'envoie en sortie. Mais comment fait-elle pour construire le dictionnaire ? Réfléchissons à la manière dont la compression procède. Lorsqu'elle envoie en sortie un indice au cours de l'étape 3C, cela signifie qu'elle a déterminé que, bien que la chaîne *s* soit présente dans le dictionnaire, la chaîne *s c* ne l'est pas. Elle envoie en sortie l'indice de *s* dans le

1. Nous supposons que les entiers sont représentés de manière standard, c'est-à-dire sur un, deux, quatre ou huit octets. En théorie, nous pourrions représenter les indices jusqu'à 230 007 en utilisant uniquement trois octets, ce qui donnerait une sortie de 689 259 octets.

Copyright © 2013 Dunod.

dictionnaire, y ajoute $s\ c$ et démarre la construction d'une nouvelle chaîne initialement fixée à $c$. La décompression doit reproduire ce comportement. Pour chaque indice lu depuis l'entrée, elle envoie en sortie la chaîne $s$ qui se trouve à cet indice dans le dictionnaire. Elle sait également qu'au moment où la compression a envoyé en sortie l'indice de $s$, la chaîne $s\ c$, où $c$ est le caractère qui vient immédiatement après $s$, ne se trouvait pas dans le dictionnaire. La décompression sait que la chaîne $s\ c$ a été ajoutée au dictionnaire, et elle doit également finir par le faire. Elle ne peut pas encore insérer $s\ c$ car elle n'a pas encore rencontré le caractère $c$. Il vient en tant que premier caractère de la chaîne suivante, que la décompression doit envoyer en sortie. Mais cette procédure n'a pas encore cette prochaine chaîne. Par conséquent, elle doit conserver la trace des deux chaînes consécutives qu'elle produit en sortie. Si la procédure de décompression envoie les chaînes *X* et *Y*, dans cet ordre, elle concatène le premier caractère de *Y* à *X* et ajoute la chaîne résultante au dictionnaire.

Prenons un exemple, avec en référence la table de la page 176, qui montre comment la compression opère sur `TATAGATCTTAATATA`. Au cours de la onzième itération, la compression envoie sur la sortie l'indice 256 pour la chaîne `TA` et ajoute la chaîne `TAA` au dictionnaire. En effet, à ce moment-là, la procédure de compression a déjà $s$ =`TA` dans le dictionnaire, alors que $s\ c$ =`TAA` en est absente. Le dernier `A` démarre la prochaine chaîne produite par la compression, `AT` (à l'indice 257), lors de la treizième itération. Par conséquent, lorsque la décompression voit les indices 256 et 257, elle doit générer `TA` et doit également mémoriser cette chaîne afin qu'au moment où elle enverra `AT` sur la sortie, elle puisse concaténer le `A` de `AT` avec `TA` et insérer la chaîne résultante, `TAA`, dans le dictionnaire.

En de rares occasions, l'indice suivant lu sur l'entrée fournie à la procédure de décompression correspond à une chaîne qui ne se trouve pas encore dans le dictionnaire. Cette situation est si rare que, lors de la décompression de *Moby Dick*, elle ne s'est produite qu'avec 15 des 229 753 indices. Elle survient lorsque l'indice généré par la procédure de compression correspond à la dernière chaîne ajoutée au dictionnaire. Ce cas se rencontre uniquement lorsque la chaîne à cet indice commence et se termine par le même caractère. En effet, rappelons que la compression envoie en sortie l'indice d'une chaîne $s$ uniquement lorsqu'elle trouve $s$ dans le dictionnaire mais pas $s\ c$, qu'elle y ajoute $s\ c$, disons à l'indice $i$, et qu'elle commence une nouvelle chaîne $s$ initialement fixée à $c$. Si l'indice suivant émis par la compression doit être $i$, alors la chaîne à l'indice $i$ dans le dictionnaire doit commencer par $c$, mais nous venons de voir que cette chaîne est $s\ c$. Par conséquent, si l'indice suivant sur l'entrée fournie à la décompression correspond à une entrée qui n'est pas encore dans le dictionnaire, cette procédure peut envoyer en sortie la dernière chaîne ajoutée au dictionnaire, concaténée avec le premier caractère de cette chaîne, et insérer celle-ci dans le dictionnaire.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Puisque de tels cas sont si rares, un exemple sera nécessairement contraint. La chaîne TATATAT en est un. La compression réalise les opérations suivantes : envoyer l'indice 84 (T) sur la sortie et insérer TA à l'indice 256, sortir l'indice 65 (A) et insérer AT à l'indice 257, sortir l'indice 256 (TA) et ajouter TAT à l'indice 258, et enfin produire l'indice 258 (TAT – la chaîne qui vient d'être ajoutée). La procédure de décompression, jusqu'à la lecture de l'indice 258, prend la dernière chaîne qu'elle a envoyée en sortie, TA, lui ajoute le premier caractère de cette chaîne, T, envoie en sortie la chaîne TAT résultante et insère celle-ci dans le dictionnaire.

Bien que cette situation rare se produise uniquement lorsque la chaîne commence et se termine par le même caractère, elle n'arrive pas chaque fois que la chaîne possède cette caractéristique. Par exemple, lors de la compression de *Moby Dick*, la chaîne dont l'indice a été envoyé en sortie avait le même caractère de début et de fin à 11 376 reprises (à peine 5 % des cas) sans être la dernière chaîne insérée dans le dictionnaire.

Voici la procédure DÉCOMPRESSION-LZW, qui précise toutes ces opérations.

*Procédure* DÉCOMPRESSION-LZW(*indices*)

*Entrée* : *indices :* une séquence d'indices dans un dictionnaire créé par COMPRESSION-LZW.

*Sortie* : Le texte fourni en entrée de COMPRESSION-LZW.

1. Pour chaque caractère *c* du jeu de caractères ASCII :
   A. Insérer *c* dans le dictionnaire à l'indice qui correspond au code numérique ASCII de *c*.
2. Fixer *courant* au premier indice de *indices*.
3. Envoyer en sortie la chaîne qui se trouve à l'indice *courant* dans le dictionnaire.
4. Tant que *indices* n'est pas vide, faire la chose suivante :
   A. Fixer *précédent* à courant.
   B. Prendre l'indice suivant de *indices* et l'affecter à *courant*.
   C. Si le dictionnaire comprend une entrée pour l'indice donné par *courant*, faire la chose suivante :
      i. Fixer *s* à la chaîne qui se trouve à l'indice *courant* dans le dictionnaire.
      ii. Envoyer la chaîne *s* en sortie.
      iii. Insérer, dans la prochaine entrée vacante du dictionnaire, la chaîne qui se trouve dans le dictionnaire à l'entrée d'indice *précédent*, concaténée avec le premier caractère de *s*.

Copyright © 2013 Dunod.

D. Sinon (le dictionnaire ne contient pas encore une entrée à l'indice *courant*), faire la chose suivante :
   i. Fixer *s* à la chaîne qui se trouve à l'indice *précédent* dans le dictionnaire, concaténée avec le premier caractère de cette entrée dans le dictionnaire.
   ii. Envoyer la chaîne *s* en sortie.
   iii. Insérer, dans la prochaine entrée vacante du dictionnaire, la chaîne *s*.

Le tableau suivant montre ce qui se passe lors de chaque itération de la boucle de l'étape 4. Nous supposons que la procédure a reçu en entrée les indices de la colonne de sortie du tableau de la page 176. Les chaînes qui correspondent aux indices *précédent* et *courant* dans le dictionnaire sont envoyées en sortie lors d'itérations consécutives, et les valeurs données pour *précédent* et *courant* dans chaque itération sont celles après l'étape 4B.

| Itération | *précédent* | *courant* | Sortie (*s*) | Nouvelle chaîne dans le dictionnaire |
|---|---|---|---|---|
| Étapes 2, 3 | | 84 | T | |
| 1 | 84 | 65 | A | 256 : TA |
| 2 | 65 | 256 | TA | 257 : AT |
| 3 | 256 | 71 | G | 258 : TAG |
| 4 | 71 | 257 | AT | 259 : GA |
| 5 | 257 | 67 | C | 260 : ATC |
| 6 | 67 | 84 | T | 261 : CT |
| 7 | 84 | 256 | TA | 262 : TT |
| 8 | 256 | 257 | AT | 263 : TAA |
| 9 | 257 | 264 | ATA | 264 : ATA |

À l'exception de la dernière itération, l'indice lu correspond à une entrée dans le dictionnaire. L'étape 4D est donc exécutée uniquement pour la dernière itération. Notez que le dictionnaire construit par DÉCOMPRESSION-LZW est identique à celui généré par COMPRESSION-LZW.

Nous n'avons pas encore expliqué comment rechercher des informations dans les dictionnaires utilisés par les procédures COMPRESSION-LZW et DÉCOMPRESSION-LZW. Pour cette dernière, la méthode est simple : il suffit de conserver le dernier indice utilisé dans le dictionnaire et, si l'indice donné par *courant* est inférieur ou égal au dernier indice utilisé, alors la chaîne se

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

trouve dans le dictionnaire. Pour la procédure COMPRESSION-LZW, la tâche est plus ardue : étant donné une chaîne, nous devons déterminer si elle se trouve dans le dictionnaire et, dans l'affirmative, à quel indice. Nous pourrions évidemment effectuer une recherche linéaire dans le dictionnaire, mais si celui-ci comprend $n$ éléments, chaque recherche se fera en un temps $O(n)$. Nous pouvons faire mieux que cela en utilisant une structure de données parmi deux possibles, mais nous n'entrerons pas dans les détails. La première est appelée **arbre à base** et est comparable à l'arbre binaire utilisé pour le codage de Huffman, excepté que chaque nœud peut avoir un nombre quelconque d'enfants, pas uniquement deux, et que les libellés des arêtes sont les caractères ASCII. La seconde structure de données est une **table de hachage** qui permet de trouver des chaînes dans le dictionnaire avec une rapidité moyenne élevée.

## Améliorations de la compression LZW

Nous l'avons indiqué, la compression du texte de *Moby Dick* avec la méthode LZW n'est pas particulièrement impressionnante. Une partie du problème vient du dictionnaire volumineux. Avec ses 230 007 entrées, chaque indice nécessite au moins 4 octets. Par conséquent, la sortie des 229 753 indices qui correspondent à la version compressée occupe une taille quatre fois plus importante, c'est-à-dire 919 012 octets. Nous pouvons remarquer deux propriétés sur les indices produits par la compression LZW. Premièrement, nombres d'entre eux sont de faible valeur et leur représentation sur 32 bits affiche donc plusieurs zéros de tête. Deuxièmement, certains indices seront plus fréquents que d'autres.

Lorsque ces deux propriétés sont constatées, le codage de Huffman va probablement donner de bons résultats. J'ai modifié le programme correspondant pour qu'il traite des entiers de 4 octets à la place des caractères et l'ai appliqué à la version de *Moby Dick* compressée selon la méthode LZW. Le fichier résultant occupe uniquement 460 971 octets, c'est-à-dire 38,61 % de la taille d'origine (1 193 826 octets). Le résultat est donc meilleur que celui obtenu avec le codage de Huffman seul. Notez cependant que la taille du codage de Huffman n'est pas comprise dans ce chiffre. De plus, tout comme la compression implique deux étapes – compression du texte avec LZW et compression des indices résultants avec le codage de Huffman –, la décompression se fait également en deux phases : décompression avec le codage de Huffman, suivie d'une décompression LZW.

D'autres améliorations de la compression de LZW visent à réduire le nombre de bits nécessaires au stockage des indices générés. Puisqu'ils sont nombreux à avoir une valeur faible, une solution consiste à employer moins de bits pour les petits nombres et à réserver les deux premiers, par exemple, pour indiquer le nombre de bits requis par l'indice.

Copyright © 2013 Dunod.

Voici un principe :

- Si les deux premiers bits sont 00, alors l'indice se trouve dans la plage 0 à 63 ($2^6 - 1$) et a besoin de six bits supplémentaires, c'est-à-dire un octet en tout.
- Si les deux premiers bits sont 01, alors l'indice se trouve dans la plage 64 ($2^6$) à 16 383 ($2^{14} - 1$) et a besoin de quatorze bits supplémentaires, c'est-à-dire deux octets en tout.
- Si les deux premiers bits sont 10, alors l'indice se trouve dans la plage 16 384 ($2^{14}$) à 4 194 303 ($2^{22} - 1$) et a besoin de vingt-deux bits supplémentaires, c'est-à-dire trois octets en tout.
- Enfin, si les deux premiers bits sont 11, alors l'indice se trouve dans la plage 4 194 304 ($2^{22}$) à 1 073 741 823 ($2^{30} - 1$) et a besoin de trente bits supplémentaires, c'est-à-dire quatre octets en tout.

Dans deux autres approches, les indices produits par la compression ont tous la même taille, car la taille du dictionnaire est limitée. Dans l'une d'elles, après que le dictionnaire a atteint la taille maximale, aucune autre entrée n'est ajoutée. Dans l'autre, le dictionnaire est vidé (à l'exception des 256 premières entrées) et la procédure de remplissage du dictionnaire reprend à partir du point du texte où le dictionnaire a été vidé. Quelle que soit la méthode retenue, la décompression doit fonctionner de manière comparable à la compression.

## Lectures complémentaires

L'ouvrage de Salomon [Sal08], particulièrement clair et concis, couvre une large diversité de techniques de compression. Celui de Storer [Sto88], publié 20 ans auparavant, est un classique du domaine. La section 16.3 du CLRS [CLRS09] plonge dans les codages de Huffman, avec un certain niveau de détail, mais elle ne prouve pas qu'ils sont les meilleurs codages préfixes possibles.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Copyright © 2013 Dunod.

# COMPLEXITÉ 10

Lorsque j'achète des biens matériels sur Internet, le marchand doit les livrer à mon domicile. La plupart du temps, il utilise une société de transport de colis. Je ne nommerai pas la société à qui la livraison de mes achats a le plus souvent été confiée, mais je peux dire que des camions marron se sont arrêtés de temps à autre devant ma porte.

## CAMIONS MARRON

Cette société de transport gère 91 000 camions marron aux États-Unis, ainsi que de nombreux autres dans le monde entier. Au moins cinq jours par semaine, chaque camion part d'un entrepôt et y revient. Pendant son périple, il livre des paquets à différentes adresses résidentielles et commerciales. La société a tout intérêt à réduire les coûts induits par les points de livraison assurés par chaque camion. Par exemple, j'ai consulté une source en ligne qui prétendait qu'en choisissant des itinéraires avec un nombre de virages à gauche inférieur, la société pouvait abaisser la distance totale parcourue par ses véhicules d'environ 750 000 kilomètres sur une période de 18 mois. Cela correspond à une économie d'environ 200 000 litres de carburant, sans compter la baisse des émissions de $CO_2$ d'environ 500 tonnes.

Comment la société réduit-elle le coût quotidien de chaque camion ? Supposons qu'un camion donné doive livrer des paquets à $n$ destinataires au cours d'une journée. En ajoutant le dépôt, le camion doit se rendre en $n + 1$ lieux. Pour chacun de ces $n + 1$ endroits, la société peut calculer le coût d'envoi du camion depuis ce lieu vers chacun des $n$ autres destinations. Elle génère donc une table de $(n + 1) \times (n + 1)$ entrées qui indiquent le coût pour aller d'une adresse à une autre. Les entrées sur la diagonale n'ont pas d'intérêt car la $i$-ème ligne et la $i$-ème colonne correspondent au même lieu. La société souhaite déterminer l'itinéraire qui part et qui revient au dépôt, en passant une seule fois par l'ensemble des $n$ autres endroits, avec un coût total le plus faible possible.

Il est possible d'écrire un programme informatique qui résout ce problème. En effet, si nous prenons un itinéraire précis et si nous connaissons l'ordre des arrêts en chemin, alors il suffit d'examiner la table des coûts

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

d'un lieu à un autre et d'en effectuer la somme. Nous devons donc simplement énumérer tous les itinéraires possibles et conserver celui qui présente le coût le plus faible. Le nombre d'itinéraires possibles est fini et le programme finira par se terminer et donner la réponse. Un tel programme ne semble pas très complexe à écrire.

Il n'est évidemment pas complexe à écrire.

En revanche, son exécution est complexe.

La difficulté vient du fait que le nombre d'itinéraires qui passent par $n$ lieux est extrêmement grand : $n!$ (factorielle $n$). En effet, le camion part du dépôt. À partir de là, n'importe lequel des $n$ autres lieux peut être le premier arrêt. À partir de ce premier arrêt, n'importe lequel des $n-1$ autres lieux peut constituer le deuxième arrêt. Il y a donc $n \cdot (n-1)$ combinaisons possibles pour les deux premiers arrêts dans l'ordre. Après que nous avons défini les deux premiers arrêts, n'importe lequel des $n-2$ autres lieux peut être le troisième arrêt. Cela donne $n \cdot (n-1) \cdot (n-2)$ ordres possibles pour les trois premiers arrêts. En poursuivant ce raisonnement aux $n$ adresses de livraison, le nombre d'ordres possibles est égal à $n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$, c'est-à-dire $n!$.

Rappelons que $n!$ croît plus rapidement qu'une fonction exponentielle. Au Chapitre 8, nous avons indiqué que 10! est égal à 3 628 800. Pour un ordinateur, ce nombre n'est pas très grand. En revanche, les camions marron livrent des paquets à plus de dix adresses chaque jour. Supposons que vingt clients soient visités quotidiennement. (Aux États-Unis, la société confie environ 170 paquets à chaque camion et, même en supposant que certains clients en attendent plusieurs, vingt arrêts par jour semblent une estimation faible.) Avec vingt arrêts, le programme informatique doit identifier les 20! ordres possibles, en sachant que 20! est égal à 2 432 902 008 176 640 000. Si les ordinateurs de l'entreprise peuvent énumérer et évaluer 1 000 milliards d'ordres par seconde, il faudra 28 jours pour traiter l'ensemble. Et cela ne correspond qu'à une seule journée de livraison pour un seul des 91 000 camions.

Avec cette approche, si la société devait acquérir et exploiter la puissance informatique nécessaire à la recherche des itinéraires quotidiens les moins onéreux pour tous les camions, les coûts informatiques engloutiraient rapidement les gains obtenus par le choix des itinéraires les plus efficaces. Cette idée d'énumérer tous les itinéraires possibles pour ne garder que le meilleur n'est tout simplement pas envisageable, même si elle semble mathématiquement intéressante. Existe-t-il une meilleure manière de trouver l'itinéraire financièrement le plus efficace pour chaque camion ?

Personne ne le sait. (Ou si quelqu'un le sait, il ne l'a pas dit.) Personne n'a trouvé une meilleure solution et personne n'a encore prouvé qu'il n'existe pas de meilleure solution. Que de frustration !

Copyright © 2013 Dunod.

La frustration est même sans doute plus grande que vous ne l'imaginez. Le problème de recherche des itinéraires les moins onéreux pour les camions marron est plus connu comme le *problème du voyageur de commerce*. Il se nomme ainsi, car dans sa formulation d'origine, un représentant doit se rendre dans $n$ villes, en partant et en arrivant dans la même ville, tout en empruntant l'itinéraire le plus court possible. Aucun algorithme qui s'exécute en un temps $O(n^c)$, pour n'importe quelle constante $c$, n'a jamais été trouvé pour ce problème. Nous ne connaissons aucun algorithme qui, à partir des distances qui séparent $n$ villes, permet de trouver le meilleur ordre possible de visite de ces $n$ villes en un temps $O(n^{100})$, $O(n^{1000})$ ou même $O(n^{1000000})$.

La situation est pire encore. De nombreux problèmes, des milliers, partagent cette caractéristique : pour une entrée de taille $n$, nous ne connaissons aucun algorithme qui s'exécute en un temps $O(n^c)$ pour n'importe quelle constante $c$, et personne n'a pu prouver qu'un tel algorithme n'existe pas. Ces problèmes se rencontrent dans divers domaines, notamment la logique, les graphes, l'arithmétique et la planification.

Augmentons encore le degré de frustration avec le fait incroyable suivant : s'il existait un algorithme qui s'exécutait en un temps $O(n^c)$ pour un quelconque de ces problèmes, avec $c$ constant, alors il existerait un algorithme qui s'exécuterait en un temps $O(n^c)$ pour l'ensemble d'entre eux. Ces problèmes sont dits **NP-complets**. Un algorithme qui s'exécute en un temps $O(n^c)$ pour une entrée de taille $n$, avec $c$ constant, est un **algorithme à temps polynomial**, car $n^c$, avec un coefficient, serait le terme le plus significatif du temps d'exécution. Nous ne connaissons aucun algorithme à temps polynomial pour n'importe quel problème NP-complet, mais personne n'a prouvé qu'il est impossible de résoudre un problème NP-complet en un temps polynomial.

Et les causes de frustration ne s'arrêtent pas là : de nombreux problèmes NP-complets sont quasiment identiques à des problèmes que nous savons résoudre en un temps polynomial. Les différences sont minimes. Par exemple, nous avons vu au Chapitre 6 que l'algorithme de Bellman-Ford permet de trouver en un temps $\Theta(nm)$ les plus courts chemins à partir d'une origine unique dans un graphe orienté de $n$ sommets et $m$ arcs, même si le graphe présente des arcs de poids négatif. Avec un graphe représenté par des listes d'adjacence, alors la taille de l'entrée est $\Theta(n + m)$. Supposons que $m \geq n$. La taille de l'entrée est alors $\Theta(m)$ et $nm \leq m^2$. L'algorithme de Bellman-Ford affiche donc un temps d'exécution polynomial en fonction de la taille de l'entrée. (Nous pouvons obtenir le même résultat avec $n > m$.) Trouver les *plus courts* chemins ne pose donc aucune difficulté. Vous pourriez en revanche être surpris d'apprendre que trouver un *plus long* chemin acyclique (c'est-à-dire un plus long chemin sans circuit) entre deux sommets est NP-complet. En réalité, simplement déterminer si un

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

graphe comprend un chemin sans circuit avec au moins un nombre d'arcs précisé est NP-complet.

Prenons un autre exemple de problèmes connexes dont l'un est facile à résoudre et l'autre est NP-complet : la tournée eulérienne et les circuits hamiltoniens. Ces deux problèmes concernent la recherche de chemins dans un graphe non orienté connexe. Dans un **graphe non orienté**, les arêtes n'ont pas de sens. Autrement dit, $(u,v)$ et $(v,u)$ représentent la même arête. Nous disons que l'arête $(u,v)$ est *incidente* aux sommets $u$ et $v$. Un **graphe connexe** possède un chemin entre chaque couple de sommets. Une **tournée eulérienne**[1] part et arrive au même sommet, et passe par chaque arête une seule fois, même si elle peut rencontrer chaque sommet à plusieurs reprises. Un **circuit hamiltonien**[2] part et arrive au même sommet, et passe une seule fois par chaque sommet (à l'exception du sommet de départ et d'arrivée). Pour déterminer si un graphe non orienté connexe comprend une tournée eulérienne, nous employons un algorithme remarquablement simple : nous déterminons le **degré** de chaque sommet, c'est-à-dire le nombre d'arêtes qui lui sont incidentes. Le graphe possède une tournée eulérienne si et seulement si le degré de chaque sommet est pair. En revanche, pour déterminer si un graphe non orienté connexe comprend un circuit hamiltonien, nous sommes face à un problème NP-complet. Notez que la question n'est pas de déterminer l'ordre des sommets dans un circuit hamiltonien dans le graphe, mais simplement s'il est possible de construire un circuit hamiltonien dans ce graphe.

Les problèmes NP-complets sont plus fréquents qu'on pourrait le penser et c'est pourquoi nous les abordons dans cet ouvrage. Si vous essayez de trouver un algorithme à temps polynomial pour un problème qui se révèle NP-complet, vous allez probablement au-devant d'une grande déception (mais consultez la section qui présente quelques idées, page 214). Le concept de problèmes NP-complets existe depuis le début des années 1970 et, bien avant cela, des personnes avaient déjà tenté de résoudre des problèmes qui se sont révélés ensuite NP-complets, comme celui du voyageur de commerce. Aujourd'hui, nous ne savons pas si un algorithme à temps polynomial existe pour un problème NP-complet, pas plus que nous savons si un tel algorithme ne peut pas exister. De nombreux brillants informaticiens ont passé des années sur cette question, sans la résoudre. Je ne prétends pas que vous ne pourrez pas trouver un algorithme à temps polynomial pour un problème NP-complet, mais simplement que les probabilités jouent contre vous.

Copyright © 2013 Dunod.

1. Ainsi nommée car le mathématicien Leonhard Euler a prouvé en 1736 qu'il était impossible d'effectuer le tour de la ville prussienne de Königsberg en franchissant ses sept ponts une seule fois et en revenant au point de départ.
2. Ce nom rend honneur à W. R. Hamilton qui, en 1856, a décrit un jeu mathématique sur un dodécaèdre. La règle est que le premier joueur plante des épingles sur cinq sommets consécutifs et que le second complète le chemin pour former un circuit comprenant tous les sommets.

# Classes de complexité N et NP, et NP-complétude

Dans les chapitres précédents, nous étions intéressés par la différence entre des temps d'exécution, par exemple entre $O(n^2)$ et $O(n \lg n)$. Dans ce chapitre, nous serons contents si un algorithme s'exécute en un temps polynomial. La différence entre $O(n^2)$ et $O(n \lg n)$ a alors peu d'importance. Les informaticiens considèrent généralement que les problèmes résolubles par des algorithmes au temps d'exécution polynomial sont « faciles » à aborder. S'il existe un algorithme à temps polynomial pour un problème, alors nous disons que ce problème appartient à la **classe P**.

Vous vous demandez peut-être comment nous pouvons considérer comme facile un problème qui nécessite un temps d'exécution $\Theta(n^{100})$. Pour une entrée de taille $n = 10$, $10^{100}$ semble plutôt un très grand nombre. C'est effectivement le cas, et la quantité $10^{100}$ est un gogol (terme à l'origine du nom « Google »). Heureusement, nous n'étudions pas d'algorithme qui prend un temps $\Theta(n^{100})$. En pratique, les problèmes de classe P que nous rencontrons s'exécutent plus rapidement. J'ai rarement vu un algorithme à temps polynomial dont le temps est pire que, disons, $O(n^5)$. Par ailleurs, dès qu'une personne trouve un algorithme à temps polynomial pour un problème, d'autres lui emboîtent le pas et proposent des algorithmes encore plus efficaces. Par conséquent, si quelqu'un écrivait le premier algorithme à temps polynomial pour un problème, mais que ce temps était $\Theta(n^{100})$, il y aurait de fortes chances que d'autres personnes proposeraient ensuite des algorithmes plus rapides.

Supposons à présent qu'on nous propose une solution à un problème et que nous souhaitions vérifier qu'elle est correcte. Par exemple, dans le problème du circuit hamiltonien, une solution proposée pourrait être une séquence de sommets. Pour vérifier qu'elle est correcte, nous devons contrôler que chaque sommet apparaît une seule fois dans la séquence, excepté le premier et le dernier qui doivent être identiques. Si la séquence est $<v_1, v_2, v_3, \ldots, v_n, v_1>$, alors le graphe doit avoir les arêtes $(v_1, v_2)$, $(v_2, v_3)$, $(v_3, v_4)$, …, $(v_{n-1}, v_n)$ et $(v_n, v_1)$. Nous pouvons aisément vérifier que cette solution est correcte en un temps polynomial. S'il est possible de contrôler une solution proposée à un problème en un temps polynomial fonction de la taille de l'entrée du problème, alors nous disons que ce problème appartient à la **classe NP**[1]. La solution proposée est un **certificat** et, pour que le problème rejoigne la *classe NP*, il faut que le temps de vérification du

1. Vous avez probablement supposé que P vient de « temps polynomial ». Si vous demandez d'où vient NP, sachez que son origine est dans l'expression « temps polynomial non déterministe ». Il s'agit d'une manière équivalente, mais pas aussi intuitive, de voir cette classe de problèmes.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

certificat soit une fonction polynomiale de la taille de l'entrée du problème et de celle du certificat.

Si nous sommes en mesure de résoudre un problème en un temps polynomial, alors nous pouvons certainement vérifier un certificat de ce problème en un temps polynomial. Autrement dit, chaque problème de P est automatiquement dans NP. La réciproque – chaque problème de NP est-il également dans P ? – est la question qui laisse perplexes les informaticiens depuis de nombreuses années. Nous l'appelons souvent le « problème $P = NP$ ? ».

Les problèmes NP-complets sont les plus « difficiles » de la classe NP. De façon informelle, un problème est **NP-complet** s'il satisfait deux conditions : (1) il est NP et (2) s'il existe un algorithme à temps polynomial pour ce problème, alors il existe une manière de transformer chaque problème de NP en ce problème de manière à les résoudre tous en un temps polynomial. S'il existe un algorithme à temps polynomial pour un problème NP-complet, autrement dit si tout problème NP-complet appartient à P, alors $P = NP$. Puisque les problèmes NP-complets sont les plus difficiles de la classe NP, s'il s'avère que tout problème de NP n'est pas résoluble en un temps polynomial, alors aucun des problèmes NP-complets ne l'est. Un problème est **NP-difficile** s'il satisfait la seconde condition de NP-complétude, mais il peut être ou ne pas être dans NP.

Voici quelques définitions :

- **P** : cette classe regroupe les problèmes qui peuvent être résolus en un temps polynomial fonction de la taille de leur entrée.
- **Certificat** : une proposition de solution à un problème.
- **NP** : cette classe réunit les problèmes qui peuvent être vérifiés en un temps polynomial ; autrement dit, étant donné un certificat, nous pouvons vérifier qu'il constitue une solution du problème en un temps polynomial fonction de la taille de l'entrée du problème et de la taille du certificat.
- **NP-difficile** : il s'agit d'un problème tel que s'il existe un algorithme à temps polynomial pour le résoudre, alors nous pouvons convertir n'importe quel problème de NP en ce problème de sorte que nous puissions les résoudre en un temps polynomial.
- **NP-complet** : il s'agit d'un problème NP-difficile qui appartient également à la classe NP.

## Problèmes de décision et réductions

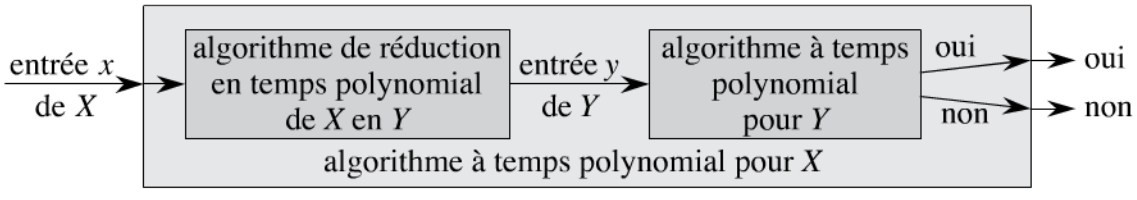
Lorsque nous parlons des classes P et NP, ou du concept de NP-complétude, nous nous limitons à des **problèmes de décision** : leur sortie est un seul bit, qui signifie « oui » ou « non ». Les problèmes de tournée eulérienne

Copyright © 2013 Dunod.

et de circuits hamiltoniens ont été énoncés sous forme de questions : « *Le graphe comprend-il une tournée eulérienne ?* » et « *Le graphe comprend-il un circuit hamiltonien ?* »

Certains problèmes sont non pas des problèmes de décision mais des problèmes d'optimisation, dans lesquels l'objectif est de trouver les meilleures solutions possibles. Heureusement, il est souvent possible de reformuler un problème d'optimisation en un problème de décision. Prenons par exemple le problème de recherche des plus courts chemins pour lequel nous avons employé l'algorithme de Bellman-Ford. Comment pouvons-nous le transformer en un problème oui/non ? Il suffit de demander « *Le graphe comprend-il un chemin entre deux sommets spécifiques avec un poids de chemin inférieur à une valeur $k$ ?* » Nous ne demandons pas les sommets ou les arcs du chemin, mais simplement si un tel chemin existe. En supposant que les poids des chemins soient des entiers, nous pouvons déterminer le poids réel du plus court chemin entre deux sommets en posant des questions oui/non. Comment ? Posons tout d'abord la question pour $k = 1$. Si la réponse est non, alors essayons avec $k = 2$. Si la réponse est toujours non, essayons avec $k = 4$. Nous répétons la question en doublant la valeur de $k$ jusqu'à ce que la réponse soit oui. Si la dernière valeur de $k$ est $k'$, alors la réponse se situe quelque part entre $k' / 2$ et $k'$. Nous recherchons la réponse oui à l'aide d'une recherche dichotomique avec un intervalle initial de $k' / 2$ à $k$. Cette approche nous donnera non pas les sommets et les arcs du plus court chemin, mais son poids.

La seconde condition pour qu'un problème soit NP-complet est que s'il existe un algorithme à temps polynomial pour ce problème, alors il est possible de convertir chaque problème de classe NP en ce problème de sorte qu'ils puissent tous être résolus en un temps polynomial. Plaçons-nous dans le cadre des problèmes de décision et réfléchissons au principe général de conversion d'un problème de décision $X$ en un autre problème de décision $Y$, de sorte que s'il existe un algorithme en temps polynomial pour $Y$, alors il en existe également un pour $X$. Une telle conversion est appelée **réduction**, car nous « réduisons » la résolution du problème $X$ à la résolution du problème $Y$. Voici l'idée :




Nous avons une entrée $x$ de taille $n$ au problème $X$. Nous transformons cette entrée en une entrée $y$ au problème $Y$ et nous le faisons en un temps polynomial fonction de $n$, disons $O(n^c)$ pour une constante $c$. La manière de transformer l'entrée $x$ en entrée $y$ doit respecter une propriété importante : si l'algorithme $Y$ arrive à « oui » pour l'entrée $y$, alors l'algorithme

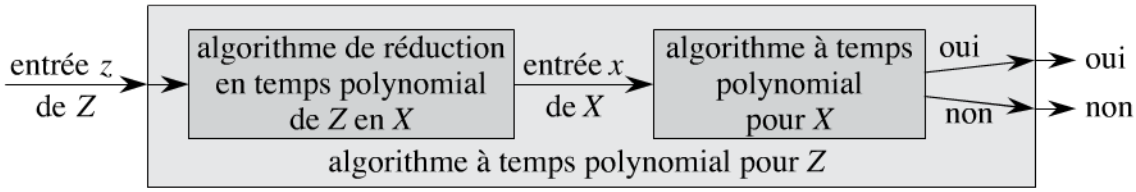
Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

$X$ doit arriver à « oui » pour l'entrée $x$, et si $Y$ conclut « non » pour $y$, alors $X$ doit conclure « non » pour $x$. Cette transformation est un **algorithme de réduction à temps polynomial**. Voyons quelle est la durée de l'intégralité de l'algorithme pour le problème $X$. L'algorithme de réduction prend un temps $O(n^c)$ et sa sortie ne peut pas être plus longue que le temps qu'il prend. Par conséquent, la taille de la sortie de l'algorithme de réduction est $O(n^c)$. Mais cette sortie est l'entrée $y$ de l'algorithme pour le problème $Y$. Puisque cet algorithme est à temps polynomial, pour une entrée de taille $m$, il s'exécute en un temps $O(m^d)$ pour une constante $d$. Dans ce cas, $m$ est $O(n^c)$ et l'algorithme pour $Y$ prend donc un temps $O((n^c)^d)$, ou $O(n^{cd})$. Puisque $c$ et $d$ sont des constantes, $cd$ l'est également, et nous constatons que le temps de l'algorithme pour $Y$ est polynomial. Le temps total de l'algorithme pour le problème $X$ est $O(n^c + n^{cd})$, ce qui en fait également un algorithme à temps polynomial.

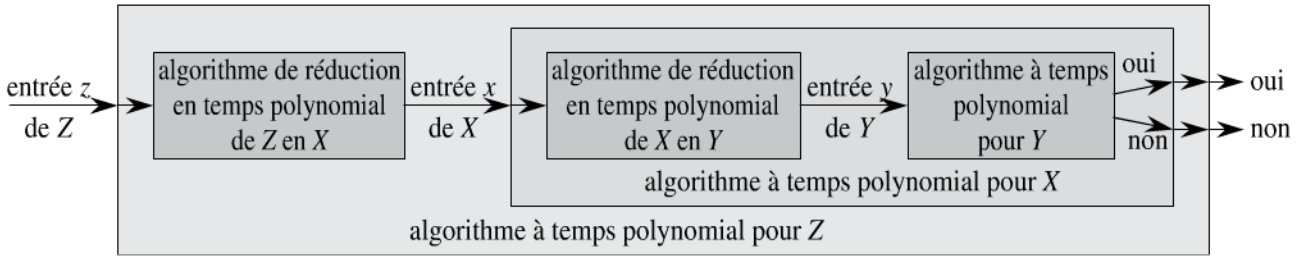
Cette approche montre que si un problème $Y$ est « facile » (résoluble en un temps polynomial), alors le problème $X$ l'est également. Mais nous allons utiliser des réductions à temps polynomial pour montrer non pas que les problèmes sont faciles mais qu'ils sont difficiles :

> Si un problème $X$ est NP-difficile et si nous pouvons le réduire en un problème $Y$ en un temps polynomial, alors le problème $Y$ est également NP-difficile.

Pourquoi cette affirmation est-elle vraie ? Supposons qu'un problème $X$ soit NP-difficile et qu'il existe un algorithme de réduction à temps polynomial pour convertir les entrées de $X$ en entrées de $Y$. Puisque $X$ est NP-difficile, il existe une manière de convertir n'importe quel problème, appelons-le $Z$, de NP en $X$ de sorte que s'il existe un algorithme en temps polynomial pour $X$, alors c'est également le cas pour $Z$. Vous savez comment cette conversion se passe, à savoir par une réduction à temps polynomial :

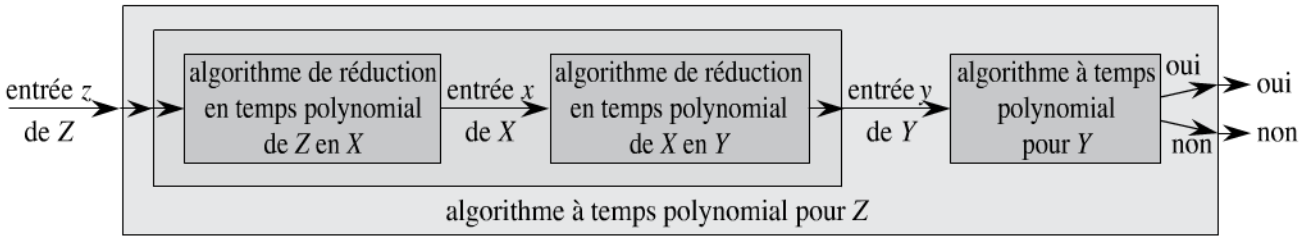



Puisque nous pouvons convertir des entrées de $X$ en entrées de $Y$ à l'aide d'une réduction à temps polynomial, nous pouvons développer $X$ comme nous l'avons fait précédemment :

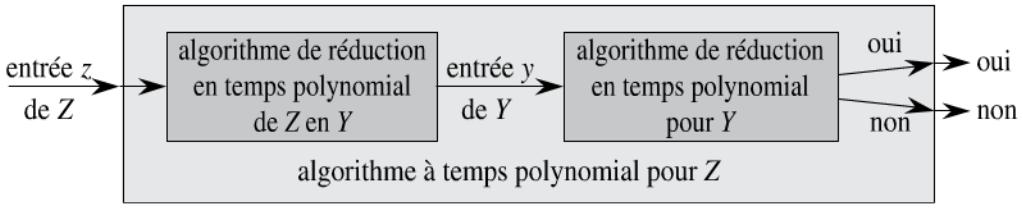



Copyright © 2013 Dunod.

Au lieu de regrouper la réduction à temps polynomial de $X$ en $Y$ et l'algorithme pour $Y$, regroupons les réductions à temps polynomial :




Si nous faisons suivre immédiatement la réduction à temps polynomial de $Z$ en $X$ par celle de $X$ en $Y$, alors nous avons une réduction à temps polynomial de $Z$ en $Y$ :




Pour être certain que les deux réductions à temps polynomial en séquence constituent une seule réduction à temps polynomial, nous allons effectuer une analyse comparable à la précédente. Supposons que l'entrée $z$ du problème $Z$ soit de taille $n$, que la réduction de $Z$ en $X$ prenne un temps $O(n^c)$, et que la réduction de $X$ en $Y$ sur une entrée de taille $m$ prenne un temps $O(m^d)$, avec $c$ et $d$ des constantes. La sortie de la réduction de $Z$ en $X$ ne peut pas être plus longue que le temps qu'il faut pour la produire et cette sortie, qui est également l'entrée $x$ de la réduction de $X$ en $Y$, a une taille $O(n^c)$. Nous savons à présent que la taille $m$ de l'entrée de la réduction de $X$ en $Y$ est $O(n^c)$. Par conséquent, le temps pris par la réduction de $X$ en $Y$ est $O((n^c)^d)$, c'est-à-dire $O(n^{cd})$. Puisque $c$ et $d$ sont des constantes, cette seconde réduction prend un temps polynomial fonction de $n$.

Par ailleurs, le temps pris lors de la dernière étape, l'algorithme à temps polynomial pour $Y$, est également polynomial en $n$. Supposons que cet algorithme prenne un temps $O(p^b)$ sur une entrée de taille $p$ et avec $b$ constant. Comme précédemment, la sortie d'une réduction ne peut pas avoir une taille qui excède le temps pris pour la produire, donc $p = O(n^{cd})$. Cela signifie que l'algorithme pour $Y$ prend un temps $O((n^{cd})^b)$, ou $O(n^{bcd})$. Puisque $b$, $c$ et $d$ sont des constantes, l'algorithme pour $Y$ prend un temps polynomial fonction de la taille d'entrée d'origine $n$. Globalement, l'algorithme pour $Z$ prend un temps $O(n^c + n^{cd} + n^{bcd})$, qui est polynomial en $n$.

Que venons-nous de voir ? Nous avons montré que si un problème $X$ est NP-difficile et que s'il existe un algorithme de réduction à temps polynomial qui transforme une entrée $x$ de $X$ en une entrée $y$ du problème $Y$, alors $Y$ est également NP-difficile. Puisque le fait que $X$ soit NP-difficile signifie que chaque problème de NP peut se réduire en $X$ en un temps polynomial,

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

nous avons pris n'importe quel problème *Z* de NP qui se réduit en *X* en un temps polynomial et avons montré qu'il se réduit également en *Y* en un temps polynomial.

Notre objectif ultime est de montrer que des problèmes sont NP-complets. Pour montrer qu'un problème *Y* est NP-complet, nous devons encore :

- montrer qu'il est dans NP, ce que nous pouvons faire en montrant qu'il existe une manière de vérifier un certificat pour *Y* en un temps polynomial, et
- prendre un autre problème *X* que nous savons NP-difficile et donner une réduction à temps polynomial de *X* en *Y*.

Il existe un petit détail supplémentaire que nous avons ignoré jusque-là : le problème mère. Nous devons partir d'un problème NP-complet *M* (le **problème mère**) dans lequel se réduit chaque problème de NP en un temps polynomial. Nous pouvons ensuite réduire *M* en un autre problème en un temps polynomial afin de montrer que l'autre problème est NP-difficile, réduire cet autre problème en un autre problème encore afin de montrer que ce dernier est NP-difficile, etc. N'oubliez pas qu'il n'existe aucune limite sur le nombre de problèmes vers lesquels nous pouvons réduire un seul problème. Par conséquent, l'arbre généalogique des problèmes NP-complets débute avec le problème mère, dont partent de nombreuses branches.

## Un problème mère

Selon les ouvrages, les problèmes mères varient. Cela n'a pas d'incidence car, après que nous avons réduit un problème mère en un autre problème, celui-ci peut servir de problèmes mère. L'un des problèmes mères le plus souvent décrits est celui de la satisfaisabilité d'une formule booléenne. Nous allons le présenter brièvement, mais nous ne prouverons pas que chaque problème de classe NP peut se réduire en ce problème en un temps polynomial. La preuve est longue et fastidieuse.

En mathématiques, « booléenne » fait référence à une logique simple dans laquelle des variables peuvent prendre uniquement les valeurs 0 ou 1 (appelées valeurs booléennes) et dont les opérateurs prennent une ou deux valeurs booléennes et produisent une valeur booléenne. Nous avons déjà rencontré le OU exclusif (XOR) au Chapitre 8. Les opérateurs booléens classiques sont ET, OU, NON, IMP (implication) et EQV (équivalence) :

- *x ET y* est égal à 1 uniquement si *x* et *y* sont tous deux égaux à 1 ; sinon (si l'un ou l'autre est égal à 0) *x ET y* est égal à 0.
- *x OU y* est égal à 0 uniquement si *x* et *y* sont tous deux égaux à 0 ; sinon (si l'un ou l'autre est égal à 1) *x OU y* est égal à 1.

Copyright © 2013 Dunod.

- *NON* $x$ correspond à l'opposé de $x$ : il vaut 0 si $x$ vaut 1, et 1 si $x$ vaut 0.
- $x$ *IMP* $y$ est égal à 0 uniquement si $x$ est égal à 1 et si $y$ est égal à 0 ; sinon (soit $x$ est égal à 0, soit $x$ et $y$ sont égaux à 1) $x$ *IMP* $y$ est égal à 1.
- $x$ *EQV* $y$ signifie « $x$ si et seulement si $y$ » et est égal à 1 uniquement si $x$ et $y$ sont égaux (tous deux à 0 ou tous deux à 1) ; si $x$ et $y$ diffèrent (l'un d'eux est égal à 0 et l'autre est égal à 1), alors $x$ *EQV* $y$ est égal à 0.

Seize opérateurs booléens peuvent prendre deux arguments, ceux décrits précédemment étant les plus courants[1]. Une **formule booléenne** est composée de variables de valeurs booléennes, d'opérateurs booléens et de parenthèses pour le regroupement.

Dans le **problème de satisfaisabilité d'une formule booléenne**, l'entrée est une formule booléenne et la question est de savoir s'il est possible d'affecter les valeurs 0 et 1 aux variables de la formule de sorte que son évaluation donne 1. Si cela est possible, nous disons que la formule est *satisfaisable*. Prenons par exemple la formule suivante :

$$((w \text{ IMP } x) \text{ OU NON } (((\text{NON } w) \text{ EQV } y) \text{ OU } z)) \text{ ET } (\text{NON } x)$$

Elle est satisfaisable avec $w = 0$, $x = 0$, $y = 1$ et $z = 1$. Voici l'évaluation de la formule avec ces valeurs :

$$\begin{aligned}((0 \text{ IMP } 0) \text{ OU NON } (((\text{NON } 0) \text{ EQV } 1) \text{ OU } 1)) \text{ ET } (\text{NON } 0)\\ &= (1 \text{ OU NON } ((1 \text{ EQV } 1) \text{ OU } 1)) \text{ ET } 1\\ &= (1 \text{ OU NON } (1 \text{ OU } 1)) \text{ ET } 1\\ &= (1 \text{ OU } 0) \text{ ET } 1\\ &= 1 \text{ ET } 1\\ &= 1\end{aligned}$$

En revanche, voici une formule simple qui n'est pas satisfaisable :

$$x \text{ ET } (\text{NON } x)$$

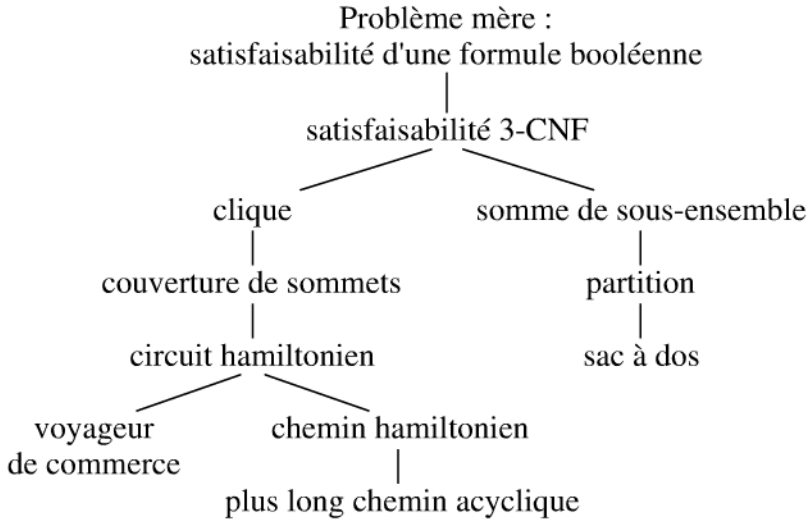
Si $x = 0$, alors son évaluation donne 0 ET 1, c'est-à-dire 0. À l'opposé, si $x = 1$, alors son évaluation donne 1 ET 0, c'est-à-dire à nouveau 0.

## Échantillon de problèmes NP-complets

En prenant le problème de satisfaisabilité d'une formule booléenne comme problème mère, voyons si nous pouvons montrer que certains problèmes sont NP-complet en utilisant des réductions à temps polynomial. Voici l'arbre généalogique des réductions que nous allons examiner :

1. Certains de ces seize opérateurs booléens binaires ne sont pas très intéressants, comme celui dont l'évaluation donne 0 quelles que soient les valeurs des opérandes.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.




Nous ne présenterons pas toutes les réductions de cet arbre généalogique, car certaines d'entre elles sont relativement longues et complexes. En revanche, nous en verrons deux dont l'intérêt est de montrer comment réduire un problème d'un domaine en un autre, comme un problème de logique (satisfaisabilité 3-CNF) en un problème de graphe (problème de la clique).

## Satisfaisabilité 3-CNF

Puisque les formules booléennes peuvent contenir n'importe quels opérateurs booléens binaires parmi les seize existants et puisque les parenthèses peuvent être employées de différentes manières, la réduction directement à partir d'un problème de satisfaisabilité d'une formule booléenne – le problème mère – est difficile. À la place, nous définirons un problème connexe qui concerne également la satisfaction de formules booléennes, mais qui pose des restrictions sur la structure de la formule donnée en entrée du problème. La réduction à partir de ce problème restreint sera beaucoup plus facile. Imposons que la formule soit constituée de ET entre des *clauses*, que chaque clause soit un OU entre trois termes et que chaque terme soit un **littéral**, c'est-à-dire une variable ou la négation d'une variable (comme NON $x$). Une telle formule booléenne est dite **sous forme normale conjonctive d'ordre 3**, ou **3-CNF** (*3-Conjunctive Normal Form*). C'est par exemple le cas de la formule booléenne suivante :

$$(w \text{ OU } (\text{NON } w) \text{ OU } (\text{NON } x)) \text{ ET } (y \text{ OU } x \text{ OU } z)$$

$$\text{ET } ((\text{NON } w) \text{ OU } (\text{NON } y) \text{ OU } (\text{NON } z))$$

Sa première clause est $(w \text{ OU } (\text{NON } w) \text{ OU } (\text{NON } x))$.

Déterminer si une formule booléenne 3-CNF est satisfaisable – le **problème de satisfaisabilité 3-CNF** – est NP-complet. Un certificat est une proposition d'affectation des valeurs 0 et 1 aux différentes variables.

Copyright © 2013 Dunod.

Vérifier un certificat n'a rien de compliqué : il suffit d'affecter les valeurs proposées aux variables et de vérifier que l'évaluation de l'expression donne 1. Pour montrer que la satisfaisabilité 3-CNF est NP-difficile, nous effectuons une réduction à partir de la satisfaisabilité d'une formule booléenne (non restreinte). Une fois encore, nous n'entrerons pas dans les détails (peu intéressants). Cela devient plus intéressant lorsque nous effectuons une réduction à partir d'un problème d'un domaine en un problème d'un domaine différent, comme nous allons le faire.

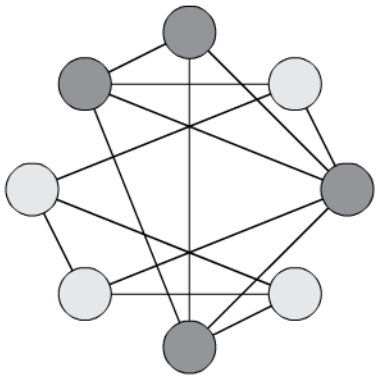
Voici un aspect frustrant de la satisfaisabilité 3-CNF : bien qu'il s'agisse d'un problème NP-complet, il existe un algorithme polynomial pour déterminer si une formule 2-CNF est satisfaisable. Une formule 2-CNF est comparable à une formule 3-CNF, excepté que chaque clause est constituée non pas de trois mais de deux littéraux. Une petite modification qui transforme un problème des plus difficiles de NP en un problème facile !

## Clique

Nous allons à présent examiner une réduction intéressante pour des problèmes dans des domaines différents : de la satisfaisabilité 3-CNF à un problème en lien avec les graphes non orientés. Dans un graphe non orienté $G$, une **clique** est un sous-ensemble $S$ de sommets tels qu'il existe une arête entre chaque couple de sommets de $S$. La **taille d'une clique** est le nombre de sommets qu'elle contient.

Comme vous pouvez l'imaginer, les cliques jouent un rôle dans la théorie des réseaux sociaux. Si nous modélisons chaque individu sous forme d'un sommet et les relations entre chaque individu sous forme d'arêtes, une clique représente un groupe d'individus qui sont tous en relation les uns avec les autres. Les cliques trouvent également des applications en bioinformatique, en ingénierie et en chimie.

Dans le **problème de la clique**, nous avons deux entrées, un graphe $G$ et un entier strictement positif $k$, et nous voulons déterminer si $G$ comprend une clique de taille $k$. Par exemple, le graphe suivant comprend une clique de taille 4, montrée par les sommets en gris foncés, et aucune autre clique de taille 4 ou supérieure.



Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

La vérification d'un certificat ne pose aucune difficulté. Le certificat correspond aux $k$ sommets qui sont censés former une clique et nous devons simplement vérifier que chacun des $k$ sommets possède une arête vers les $k-1$ autres sommets. Ce contrôle peut se faire aisément en un temps polynomial fonction de la taille du graphe. Nous savons à présent que le problème de la clique appartient à la classe NP.

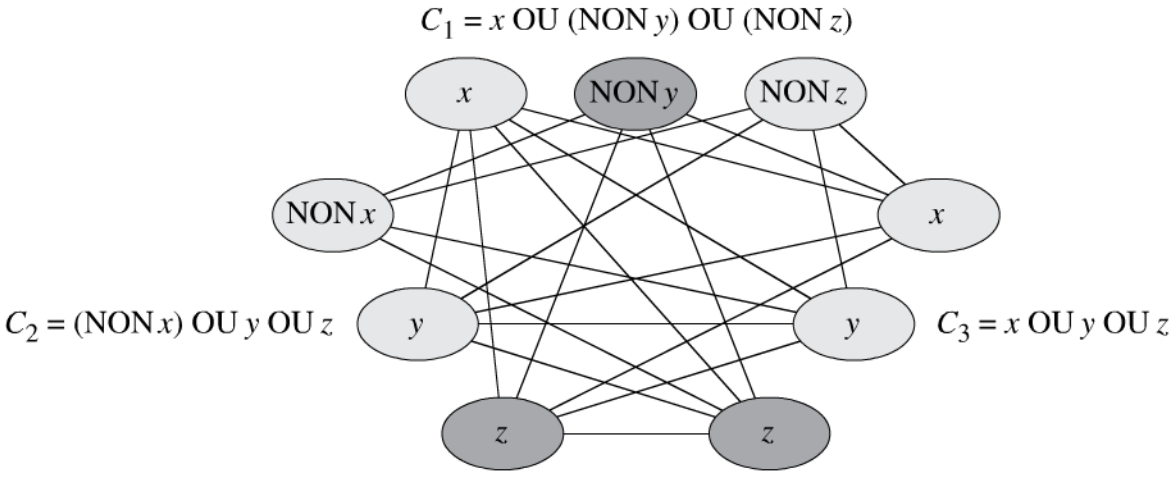
Comment un problème de satisfaction de formule booléenne peut-il se réduire en un problème de graphe ? Nous partons d'une formule 3-CNF. Supposons qu'elle soit $C_1$ ET $C_2$ ET $C_3$ ET… ET $C_k$, ou chaque $C_r$ est une clause parmi $k$. À partir de cette formule, nous pouvons construire un graphe en un temps polynomial et ce graphe possédera une clique de taille $k$ si et seulement si la formule 3-CNF est satisfaisable. Nous devons examiner trois points : la construction, une preuve que la construction se fait en un temps polynomial fonction de la taille de la formule 3-CNF, et une preuve que le graphe comprend une clique de taille $k$ si et seulement s'il est possible d'affecter des valeurs aux variables de la formule 3-CNF de sorte que son évaluation donne 1.

Pour construire un graphe à partir d'une formule 3-CNF, intéressons-nous à la $r$-ème clause, $C_r$. Elle comprend trois littéraux, que nous nommons $l_1^r$, $l_2^r$ et $l_3^r$. $C_r$ est donc $l_1^r$ OU $l_2^r$ OU $l_3^r$. Chaque littéral est soit une variable, soit la négation d'une variable. Nous créons un sommet pour chaque littéral, et, pour une clause $C_r$, nous avons donc trois sommets, $v_1^r$, $v_2^r$ et $v_3^r$. Nous ajoutons une arête entre des sommets $v_i^r$ et $v_j^s$ si les deux conditions suivantes sont satisfaites :

- $v_i^r$ et $v_j^s$ sont dans des triplets différents, autrement dit $r$ et $s$ ont des numéros de clause différents ;
- les littéraux correspondants ne sont pas des négations l'un de l'autre.

Par exemple, le graphe donné ci-après correspond à la formule 3-CNF suivante :

$$(x \text{ OU } (\text{NON } y) \text{ OU } (\text{NON } z)) \text{ ET } ((\text{NON } x) \text{ OU } y \text{ OU } z) \text{ ET } (x \text{ OU } y \text{ OU } z)$$




Copyright © 2013 Dunod.

Nous pourrons facilement déterminer que cette réduction est réalisable en un temps polynomial. Si la formule 3-CNF comprend $k$ clauses, alors elle comprend $3k$ littéraux et le graphe a $3k$ sommets. Chaque sommet possède au plus une arête vers tous les $3k - 1$ autres sommets. Le nombre d'arêtes est donc au plus égal à $3k(3k - 1)$, c'est-à-dire $9k^2 - 3k$. La taille du graphe construit est donc une fonction polynomiale de la taille de l'entrée de la formule 3-CNF, et il est facile de connaître les arêtes qui vont dans le graphe.

Enfin, nous devons montrer que le graphe construit comprend une clique de taille $k$ si et seulement si la formule 3-CNF est satisfaisable. Nous commençons par supposer qu'elle est satisfaisable et montrons que le graphe comprend une clique de taille $k$. S'il existe des affectations satisfaisantes, chaque clause $C_r$ contient au moins un littéral $l_r^i$ dont l'évaluation donne 1, et chaque tel littéral correspond à un sommet $v_i^r$ du graphe. Si nous choisissons un tel littéral pour chacune des $k$ clauses, nous obtenons un ensemble $S$ correspondant de $k$ sommets. Nous prétendons que $S$ est une clique de taille $k$. Considérons deux sommets quelconques de $S$. Ils correspondent à des littéraux dans des clauses différentes dont l'évaluation donne 1 avec les affectations satisfaisantes. Ces littéraux ne peuvent pas être des négations l'un de l'autre, car, dans le cas contraire, l'un d'eux vaudrait 1 et l'autre 0. Puisque ces littéraux ne sont pas des négations l'un de l'autre, nous avons créé une arête entre ces deux sommets au cours de la construction du graphe. Puisque les deux sommets choisis peuvent être n'importe quel couple de sommets de $S$, nous en concluons qu'il existe des arêtes entre tous les couples de sommets de $S$. Autrement dit, $S$, un ensemble de $k$ sommets, est une clique de taille $k$.

Nous raisonnons à présent dans l'autre sens : si le graphe comprend une clique $S$ de taille $k$, alors la formule 3-CNF est satisfaisable. Aucune arête du graphe ne relie des sommets du même triplet et $S$ contient donc exactement un sommet par triplet. Pour chaque sommet $v_i^r$ de $S$, nous affectons 1 au littéral $l_i^r$ correspondant dans la formule 3-CNF. Nous n'avons pas à nous inquiéter de l'affectation éventuelle d'un 1 à un littéral et à sa négation, car la clique de taille $k$ ne peut pas contenir des sommets qui correspondent à un littéral et à sa négation. Puisque chaque clause comprend un littéral dont l'évaluation donne 1, elles sont toutes satisfaites et l'intégralité de la formule 3-CNF l'est également. Si une variable ne correspond pas à des sommets de la clique, nous lui affectons une valeur arbitraire ; elle n'a pas d'influence sur la satisfaisabilité de la formule.

Dans l'exemple précédent, $y = 0$ et $z = 1$ est une affectation satisfaisante ; la valeur donnée à $x$ n'a pas d'importance. Une clique de taille 3 correspondante est constituée de tous les sommets en gris foncé, c'est-à-dire ceux qui correspondent à *NON* $y$ dans la clause $C_i$ et à $z$ dans les clauses $C_2$ et $C_3$.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Nous avons donc montré qu'il existe une réduction à temps polynomial du problème NP-complet de satisfaisabilité 3-CNF en un problème de recherche d'une clique de taille $k$. Si vous recevez une formule booléenne 3-CNF avec $k$ clauses et si vous devez trouver une affectation qui satisfait la formule, vous pouvez utiliser la construction que nous venons de décrire pour convertir en un temps polynomial la formule en un graphe non orienté et déterminer si celui-ci comprend une clique de taille $k$. Si vous êtes en mesure de déterminer en un temps polynomial que le graphe possède une clique de taille $k$, alors vous pouvez déterminer en un temps polynomial qu'il existe une affectation qui satisfait la formule 3-CNF. Puisque le problème de satisfaisabilité 3-CNF est NP-complet, déterminer si un graphe comprend une clique de taille $k$ l'est également. Par ailleurs, si vous pouvez également déterminer les sommets qui composent la clique de taille $k$, alors vous pouvez employer cette information pour trouver les valeurs à donner aux variables de la formule 3-CNF pour qu'elle soit satisfaite.

## Couverture de sommets

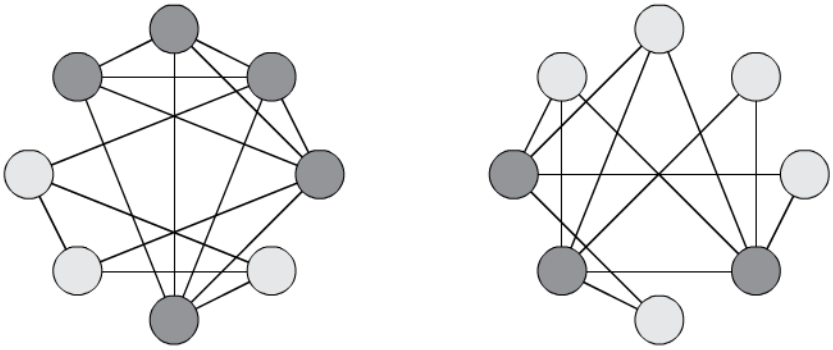
Une **couverture de sommets** dans un graphe non orienté $G$ est un sous-ensemble $S$ de sommets tel que chaque arête de $G$ est incidente à au moins un sommet de $S$. Nous disons que chaque sommet de $S$ « couvre » ses arêtes incidentes. La **taille d'une couverture de sommets** est son nombre de sommets. À l'instar du problème de la clique, le **problème de la couverture de sommets** prend en entrée un graphe non orienté $G$ et un entier strictement positif $m$. La question est de savoir si $G$ comprend une couverture de sommets de taille $m$. Ce problème de couverture de sommets trouve une application en bio-informatique. Dans une autre, nous avons un bâtiment avec des couloirs. Des caméras couvrant un angle de 360° sont positionnées aux intersections des couloirs. Nous voulons savoir si $m$ caméras permettent de surveiller l'intégralité des couloirs. Dans cet exemple, les arêtes modélisent les couloirs et les sommets correspondent aux intersections. Dans une autre application encore, la recherche des couvertures de sommets sert à concevoir des stratégies de mises en échec des attaques virales dans les réseaux informatiques.

Dans le cas du problème de couverture de sommets, un certificat est, sans surprise, une proposition de couverture de sommets. Nous pouvons facilement vérifier en un temps polynomial fonction de la taille du graphe si la couverture de sommets proposée a une taille $m$ et si elle couvre réellement toutes les arêtes, et ainsi conclure que ce problème est dans la classe NP.

L'arbre généalogique de NP-complétude donné à la page page 196 nous indique que nous réduisons le problème de la clique en un problème de couverture de sommets. Supposons que l'entrée du problème de la clique soit un graphe non orienté $G$ de $n$ sommets et un entier strictement positif $k$. En un temps polynomial, nous produisons un graphe d'entrée $\overline{G}$ pour

Copyright © 2013 Dunod.

le problème de couverture de sommets tel que $G$ comprend une clique de taille $k$ si seulement si $\overline{G}$ possède une couverture de sommets de taille $n-k$. Cette réduction est vraiment facile. Le graphe $\overline{G}$ comprend les mêmes sommets que $G$ et ses arêtes sont exactement à l'opposé de celles de $G$. Autrement dit, l'arête $(u,v)$ se trouve dans $\overline{G}$ si et seulement si $(u,v)$ n'est *pas* dans $G$. Vous aurez sans doute deviné que la couverture de sommets de taille $n-k$ dans $\overline{G}$ est constituée des sommets qui ne se trouvent *pas* dans la clique de $k$ sommets de $G$. Voici un exemple de graphes $G$ et $\overline{G}$, avec huit sommets. Les cinq sommets qui forment une clique dans $G$ et les trois autres sommets qui forment une couverture de sommets dans $\overline{G}$ sont indiqués en gris foncé :



$G$ $\overline{G}$

Vous remarquerez que chaque arête de $\overline{G}$ est incidente à au moins un sommet gris foncé.

Nous devons montrer que $G$ comprend une clique de taille $k$ si et seulement si $\overline{G}$ comprend une couverture de sommets de taille $n-k$. Commençons par supposer que $G$ présente une clique $C$ de taille $k$. Soit $S$ l'ensemble des $n-k$ sommets qui ne se trouvent pas dans $C$. Nous prétendons que chaque arête de $\overline{G}$ est incidente à au moins un sommet de $S$. Soit $(u,v)$ une arête quelconque de $\overline{G}$. Elle se trouve dans $\overline{G}$ car elle n'est pas dans $G$. Puisque $(u,v)$ n'est pas dans $G$, au moins l'un des sommets $u$ et $v$ n'est pas dans la clique $C$ de $G$, car une arête relie chaque couple de sommets de $C$. Puisqu'au moins $u$ ou $v$ n'est pas dans $C$, au moins l'un ou l'autre se trouve dans $S$. Autrement dit, cette arête $(u,v)$ est incidente à au moins un des sommets de $S$. Puisque nous avons choisi une arête $(u,v)$ quelconque de $\overline{G}$, nous en concluons que $S$ est une couverture de sommets pour $\overline{G}$.

Nous devons à présent raisonner en sens inverse. Supposons que $\overline{G}$ comprenne une couverture de sommets $S$ constituée de $n-k$ sommets. Soit $C$ l'ensemble des $k$ sommets qui ne se trouvent pas dans $S$. Chaque arête de $\overline{G}$ est incidente à un certain sommet de $S$. Autrement dit, si $(u,v)$ est une arête de $\overline{G}$, alors au moins l'un des sommets $u$ ou $v$ se trouve dans $S$. En reprenant la définition de la contraposée donnée à la page 22, nous

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

établissons que la contraposée de cette implication est que si ni $u$ ni $v$ n'est dans $S$, alors $(u,v)$ n'est pas dans $\overline{G}$, et par conséquent que $(u,v)$ est dans $G$. Autrement dit, si $u$ et $v$ sont dans $C$, alors l'arête $(u,v)$ est présente dans $G$. Puisque $u$ et $v$ forment n'importe quel couple de sommets de $C$, nous déterminons qu'il existe une arête dans $G$ entre tous les couples de sommets de $C$. Par conséquent, $C$ est une clique de taille $k$.

Nous avons donc montré que le problème NP-complet de déterminer si un graphe non orienté comprend une clique de taille $k$ peut être réduit en un temps polynomial en un problème de déterminer si un graphe non orienté comprend une couverture de sommets de taille $n - k$. Si vous recevez un graphe non orienté $G$ et devez déterminer s'il contient une clique de taille $k$, vous pouvez utiliser la construction que nous venons de décrire pour convertir en un temps polynomial $G$ en $\overline{G}$ et déterminer si $\overline{G}$ contient une couverture de sommets de $n - k$ sommets. Si vous pouvez déterminer en un temps polynomial que $\overline{G}$ comprend une couverture de sommets de taille $n - k$, alors vous pouvez déterminer en un temps polynomial que $G$ a une clique de taille $k$. Puisque le problème de la clique est NP-complet, le problème de la couverture de sommets l'est également. Par ailleurs, si vous pouvez non seulement déterminer que $\overline{G}$ a une couverture de $n - k$ sommets, mais également identifier ces sommets, alors vous pouvez utiliser cette information pour déterminer les sommets de la clique de taille $k$.

## Circuit et chemin hamiltoniens

Nous avons déjà présenté le problème de circuit hamiltonien : un graphe non orienté connexe contient-il un circuit hamiltonien (un chemin qui part et qui arrive au même sommet et qui passe par tous les autres sommets exactement une fois) ? Les applications de ce problème sont un tantinet ésotériques, mais l'arbre généalogique de NP-complétude (page 196) indique qu'il est utilisé pour démontrer que le problème du voyageur de commerce est NP-complet, qui lui se rencontre dans la pratique.

Le **problème de chemin hamiltonien** est proche. La question est de savoir si le graphe contient un chemin qui passe exactement une fois par chaque sommet, sans qu'il constitue nécessairement un circuit fermé. Ce problème est également NP-complet et nous nous en servirons à la page 205 pour montrer que le problème de plus long chemin acyclique est NP-complet.

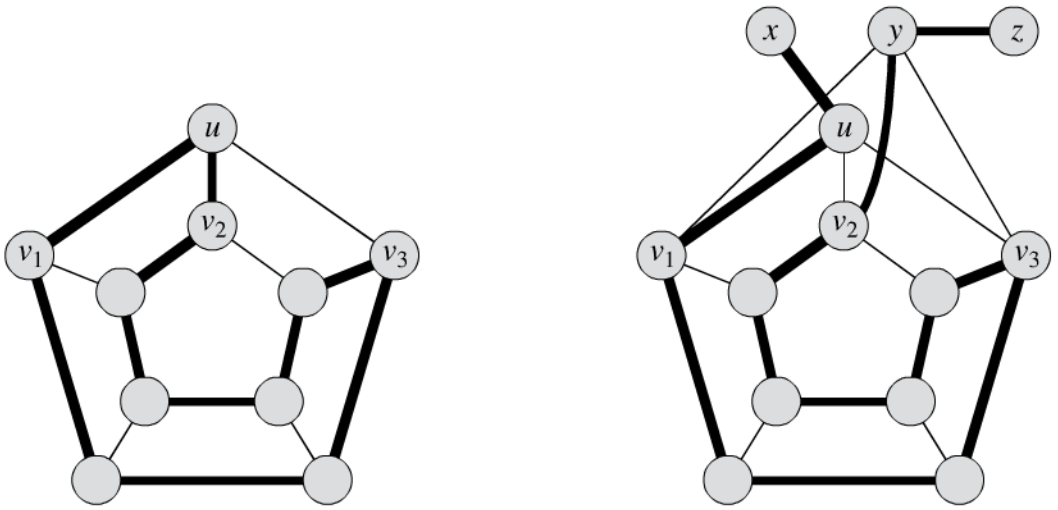
Dans les deux problèmes hamiltoniens, le certificat est évident : l'ordre des sommets dans le circuit ou le chemin hamiltonien. (Dans le cas d'un chemin hamiltonien, le premier sommet n'est pas répété à la fin.) Étant donné un certificat, nous devons seulement vérifier que chaque sommet apparaît exactement une fois dans la liste et que le graphe contient une arête entre chaque couple de sommets consécutifs, dans l'ordre. Pour le

Copyright © 2013 Dunod.

problème du circuit hamiltonien, nous devons également vérifier qu'il existe une arête entre le premier et le dernier sommet.

Nous n'entrerons pas dans les détails de la réduction à temps polynomial du problème de couverture de sommet en problème de circuit hamiltonien, qui montre que ce dernier est NP-difficile. Elle est relativement complexe et se fonde sur un **gadget**. Un gadget correspond à une partie d'un graphe qui respecte certaines propriétés. Le gadget utilisé dans la réduction présente la propriété suivante : un circuit hamiltonien dans le graphe construit par la réduction peut traverser le gadget selon une des trois seules manières possibles.

Pour réduire le problème de circuit hamiltonien en problème de chemin hamiltonien, nous partons d'un graphe non orienté connexe $G$ de $n$ sommets et nous l'utilisons pour créer un nouveau graphe non orienté connexe $G'$ avec $n+3$ sommets. Nous choisissons n'importe quel sommet $u$ de $G$ et nommons $v_1, v_2, \ldots, v_k$ ses sommets adjacents. Pour construire $G'$, nous ajoutons trois nouveaux sommets, $x$, $y$ et $z$, ainsi que les arêtes $(u,x)$ et $(y,z)$, sans oublier les arêtes $(v_1,y),(v_2,y),\ldots,(v_k,y)$ entre $y$ et tous les sommets adjacents à $u$. En voici un exemple :




$G$ $G'$

Les arêtes épaisses forment un circuit hamiltonien dans $G$ et le chemin hamiltonien correspondant dans $G'$. La réduction se fait en un temps polynomial car $G'$ comprend uniquement trois sommets de plus que $G$ et au plus $n+1$ arêtes supplémentaires.

À nouveau, il nous faut montrer que la réduction fonctionne : que $G$ possède un circuit hamiltonien si et uniquement si $G'$ comprend un chemin hamiltonien. Supposons que $G$ inclue un circuit hamiltonien. Il doit contenir une arête $(u,v_i)$ pour un sommet $v_i$ adjacent à $u$ et, par conséquent, adjacent à $y$ dans $G'$. Pour former un chemin hamiltonien dans $G$, qui va de $x$ à $z$, nous conservons toutes les arêtes du circuit hamiltonien, à l'exception de $(u,v_i)$, et ajoutons les arêtes $(u,x)$, $(v_i,y)$ et $(y,z)$. Dans

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

l'exemple précédent, $v_2$ est le sommet $v_i$ et le chemin hamiltonien retire donc l'arête $(v_2,u)$ et ajoute les arêtes $(u,x)$, $(v_2,y)$ et $(y,z)$.

Supposons à présent que $G'$ comprenne un chemin hamiltonien. Puisque les sommets $x$ et $z$ ont chacun uniquement une seule arête incidente, le chemin hamiltonien doit aller de $x$ à $z$ et il doit contenir une arête $(v_i,y)$ pour un sommet adjacent à $y$ et, par conséquent, adjacent à $u$. Pour rechercher un circuit hamiltonien dans $G$, nous retirons $x$, $y$ et $z$, ainsi que toutes leurs arêtes incidentes, et utilisons toutes les arêtes du chemin hamiltonien dans $G'$, sans oublier $(v_i,u)$.

Un dénouement comparable à celui de nos réductions précédentes se produit dans ce cas. Il existe une réduction à temps polynomial du problème NP-complet qui vise à déterminer si un graphe non orienté connexe contient un circuit hamiltonien en un problème qui consiste à déterminer si un graphe non orienté connexe contient un chemin hamiltonien. Puisque le premier est NP-complet, le dernier l'est également. Par ailleurs, en connaissant les arêtes du chemin hamiltonien, nous avons les arêtes du circuit hamiltonien.

## Voyageur de commerce

Dans le **problème du voyageur de commerce** en version décision, nous disposons d'un graphe non orienté complet avec des poids entiers positifs sur chaque arête et un entier positif $k$. Dans un **graphe complet**, il existe une arête entre chaque paire de sommets. Par conséquent, si un graphe complet est constitué de $n$ sommets, il comprend alors $n(n-1)$ arêtes. La question est de savoir si le graphe présente un circuit qui passe par tous les sommets et dont le poids total est inférieur ou égal à $k$.

Il est assez facile de montrer que ce problème se trouve dans la classe NP. Un certificat est constitué des sommets du circuit, dans l'ordre. Nous pouvons aisément vérifier en un temps polynomial si les arêtes de ce circuit passent par tous les sommets et si elles ont un poids total inférieur ou égal à $k$.

Pour montrer que le problème du voyageur de commerce est NP-difficile, nous effectuons une réduction à partir du problème de circuit hamiltonien. À partir d'un graphe $G$ en entrée du problème de circuit hamiltonien, nous construisons un graphe complet $G'$ avec les mêmes sommets que $G$. Nous fixons le poids de l'arête $(u,v)$ dans $G'$ à 0 si $(u,v)$ est une arête de $G$. Nous le fixons à 1 si l'arête $(u,v)$ n'existe pas dans $G$. Nous fixons $k$ à 0. Cette réduction se fait en un temps polynomial fonction de la taille de $G$, car elle ajoute au plus $n(n-1)$ sommets.

Pour montrer que la réduction fonctionne, nous devons montrer que $G$ possède un circuit hamiltonien si et seulement si $G'$ comprend un circuit de poids 0 qui passe par tous les sommets. L'argumentation est

Copyright © 2013 Dunod.

de nouveau facile. Supposons que $G$ comprenne un circuit hamiltonien. Dans ce cas, chaque arête du circuit se trouve dans $G$ et chaque arête a donc un poids égal à 0 dans $G'$. Par conséquent, $G'$ présente un circuit qui contient tous les sommets et dont le poids total est égal à 0. Réciproquement, supposons à présent que $G'$ possède un circuit qui contient tous les sommets et dont le poids total est égal à 0. Alors, chaque arête de ce circuit doit également se trouver dans $G$ et ce graphe comprend donc un circuit hamiltonien.

Ai-je besoin de répéter le dénouement à présent familier ?

## Plus long chemin acyclique

Dans le **problème du plus long chemin acyclique** en version décision, nous disposons d'un graphe non orienté $G$ et d'un entier $k$. La question est de savoir si $G$ contient deux sommets pour lesquels il existe entre eux un chemin acyclique de $k$ arêtes.

Une fois encore, un certificat pour le problème du plus long chemin acyclique est facile à vérifier. Il est constitué des sommets du chemin proposé, dans l'ordre. Nous pouvons vérifier en un temps polynomial que la liste comprend au moins $k+1$ sommets (car un chemin avec $k$ arêtes contient $k+1$ sommets), sans sommet en double, et qu'il existe une arête entre chaque couple de sommets consécutifs dans la liste.

Une autre réduction simple montre que ce problème est NP-difficile. Elle se fait à partir du problème de chemin hamiltonien. Étant donné un graphe $G$ de $n$ sommets en entrée du problème de chemin hamiltonien, nous avons en entrée du problème de plus long chemin acyclique le graphe $G$ non modifié et l'entier $k = n - 1$. S'il ne s'agit pas d'une réduction à temps polynomial, je veux bien être pendu.

Nous montrons que la réduction fonctionne en montrant que $G$ comprend un chemin hamiltonien si et seulement s'il possède un chemin acyclique qui contient au moins $n-1$ arêtes. Puisque le chemin hamiltonien est un chemin acyclique qui contient $n-1$ arêtes, la preuve est faite !

## Somme de sous-ensemble

Dans le **problème de somme de sous-ensemble**, l'entrée est un ensemble $S$ fini d'entiers strictement positifs, sans ordre particulier, et un entier cible $t$, également entier strictement positif. La question est de savoir s'il existe un sous-ensemble $S'$ de $S$ dont la somme des éléments est égale à $t$. Par exemple, si $S$ est l'ensemble {1,2,7,14,49,98,343,686,2409,2793, 16808,17206,117705,117993} et si $t = 138457$, alors le sous-ensemble $S' = \{1,2,7,98,343,686,2409,17206,117705\}$ est une solution. Un certificat est évidemment un sous-ensemble de $S$, que nous pouvons vérifier

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

en additionnant simplement les nombres du sous-ensemble et en contrôlant que le résultat est égal à $t$.

L'arbre généalogique de NP-complétude de la page 196 montre que le problème de somme de sous-ensemble est NP-difficile par une réduction à partir du problème de satisfaisabilité 3-CNF. Voici une autre réduction transverse aux domaines des problèmes : un problème de logique est transformé en problème d'arithmétique. Vous allez constater que la transformation est ingénieuse, mais somme toute assez évidente.

Nous partons d'une formule booléenne 3-CNF $F$ constituée de $n$ variables et de $k$ clauses. Nommons les variables $v_1, v_2, v_3, \ldots, v_n$ et les clauses $C_1, C_2, C_3, \ldots, C_k$. Chaque clause contient exactement trois littéraux (rappelez-vous que chaque littéral est soit $v_i$, soit NON $v_i$) réunis par des OU et l'intégralité de la formule $F$ est $C_1$ ET $C_2$ ET $C_3$ ET … ET $C_k$. Autrement dit, pour une affectation donnée de 0 et de 1 à chaque variable, chaque clause est satisfaite si l'évaluation de l'un de ses littéraux donne 1. L'intégralité de la formule $F$ est satisfaite uniquement si toutes ses clauses sont satisfaites.

Avant de construire l'ensemble $S$ pour le problème de somme de sous-ensemble, construisons le nombre cible $t$ à partir de la formule 3-CNF $F$, sous forme d'un entier décimal de $n + k$ chiffres. Les $k$ chiffres les moins significatifs (les $k$ chiffres de droite) de $t$ correspondent aux $k$ clauses de $F$, et chacun d'eux est 4. Les $n$ chiffres les plus significatifs de $t$ correspondent aux $n$ variables de $F$, et chacun d'eux est 1. Si la formule possède, par exemple, trois variables et quatre clauses, alors $t$ est égal à 1114444. Comme nous le verrons, s'il existe un sous-ensemble de $S$ dont la somme des éléments est égale à $t$, alors les chiffres de $t$ qui correspondent aux variables (les 1) garantiront que nous affectons une valeur à chaque variable de $F$, et les chiffres de $t$ qui correspondent aux clauses (les 4) garantiront que chaque clause de $F$ est satisfaite.

L'ensemble $S$ comprend $2n + 2k$ entiers. Il contient des entiers $x_i$ et $x_i'$ pour chacune des $n$ variables $v_i$ de la formule 3-CNF $F$, et il contient des entiers $q_j$ et $q_j'$ pour chacune des $k$ clauses $C_j$ de $F$. Nous construisons chaque entier de $S$ chiffre par chiffre, en décimal. Voyons un exemple avec $n = 3$ variables et $k = 4$ clauses. La formule 3-CNF est donc $F = C_1$ ET $C_2$ ET $C_3$ ET $C_4$. Les clauses sont les suivantes :

$C_1 = v_1$ OU (NON $v_2$) OU (NON $v_3$)

$C_2 =$(NON $v_1$) OU (NON $v_2$) OU (NON $v_3$)

$C_3 =$(NON $v_1$) OU (NON $v_2$) OU $v_3$

$C_4 = v_1$ OU $v_2$ OU $v_3$

Voici l'ensemble $S$ et la cible $t$ correspondants.

Copyright © 2013 Dunod.

| | | $v_1$ | $v_2$ | $v_3$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | = | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| $x'_1$ | = | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| $x_2$ | = | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| $x'_2$ | = | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| $x_3$ | = | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| $x'_3$ | = | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| $q_1$ | = | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $q'_1$ | = | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| $q_2$ | = | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $q'_2$ | = | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| $q_3$ | = | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $q'_3$ | = | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| $q_4$ | = | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| $q'_4$ | = | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| $t$ | = | 1 | 1 | 1 | 4 | 4 | 4 | 4 |

Notez que la somme des éléments de $S$ grisés – 1000110, 101110, 10011, 1000, 2000, 200, 10, 1 et 2 – est égale à 1114444. Nous verrons ci-après à quoi correspondent ces éléments dans la formule 3-CNF $F$.

Nous construisons les entiers de $S$ de sorte que, chiffre après chiffre, la somme de chaque colonne de la table précédente soit égale à 2 (pour les $n$ colonnes de gauche) ou à 6 (pour les $k$ colonnes de droite). Notez que l'addition des éléments de $S$ ne produit aucune retenue et nous pouvons travailler avec les nombres chiffre par chiffre.

Dans la table, chaque rangée est libellée par un élément de $S$. Les $2n$ premières rangées correspondent aux $n$ variables de la formule 3-CNF. Les $2k$ dernières rangées servent de « marge », mais nous y reviendrons plus loin. Les rangées libellées par des éléments $x_i$ et $x'_i$ correspondent respectivement aux occurrences des littéraux $v_i$ et NON $v_i$ dans $F$. Nous dirons que ces rangées « sont » les littéraux, en sachant que cela signifie qu'ils correspondent aux littéraux. L'objectif est d'inclure dans le sous-ensemble $S'$ exactement $n$ des $2n$ premières rangées – une seule pour chaque couple $x_i$, $x'_i$ – qui correspondront à une affectation satisfaisante pour la formule 3-CNF $F$. Puisque les rangées choisies à partir des littéraux doivent avoir une somme égale à 1 dans chacune des $n$ colonnes de gauche, nous sommes certains que, pour chaque variable $v_i$ de la formule 3-CNF, nous incluons dans $S'$ une rangée de $x_i$ ou de $x'_i$, non les deux. Les $k$ colonnes de droite garantissent que les rangées incluses dans $S'$ sont des littéraux qui satisfont chacune des clauses de la formule 3-CNF.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Concentrons-nous pour le moment sur les $n$ colonnes de gauche, dont les libellés sont les variables $v_1, v_2, \ldots, v_n$. Pour une variable $v_i$ donnée, $x_i$ et $x'_i$ possèdent un 1 dans le chiffre correspondant de $v_i$. Ils ont un 0 dans tous les autres emplacements de chiffres qui correspondent aux autres variables. Par exemple, les trois chiffres de gauche de $x_2$ et de $x'_2$ sont 010. Les chiffres des $2k$ dernières rangés des $n$ colonnes de gauche sont des 0. Puisque la cible $t$ possède un 1 dans chacun des emplacements de variables, $x_i$ ou $x'_i$ doit se trouver dans le sous-ensemble $S'$ pour contribuer à la somme. S'il s'agit de $x_i$, $v_i$ est fixé à 1. S'il s'agit de $x'_i$, $v_i$ est fixé à 0.

Passons à présent aux $k$ colonnes de droite, qui correspondent aux clauses. Comme nous allons le voir, ces colonnes garantissent que chaque clause est satisfaite. Si le littéral $v_i$ apparaît dans une clause $C_j$, alors $x_i$ a un 1 dans la colonne de $C_j$ ; si le littéral NON $v_i$ apparaît dans une clause $C_j$, alors $x'_i$ a un 1 dans la colonne de $C_j$. Puisque chaque clause d'une formule 3-CNF comprend exactement trois littéraux distincts, la colonne de chaque clause doit comprendre exactement trois 1 dans les rangées $x_i$ et $x'_i$. Pour une clause $C_j$ donnée, les rangées parmi les $2n$ premières qui sont incluses dans $S'$ correspondent à la satisfaction de 0, 1, 2 ou 3 des littéraux de $C_j$, et ces rangées contribuent donc à hauteur de 0, 1, 2 ou 3 dans la somme de la colonne de $C_j$.

Toutefois, le chiffre cible de chaque clause est 4 et c'est là que les éléments de « marge » $q_j$ et $q'_j$, pour $j = 1,2,3,\ldots,k$, entrent en scène. Ils garantissent que pour chaque clause, le sous-ensemble $S'$ comprend un littéral de la clause (un $x_i$ ou $x'_i$ qui a un 1 dans la colonne de cette clause). La rangée de $q_j$ possède un 1 dans la colonne de la cause $C_j$ et un 0 partout ailleurs, et la rangée de $q'_j$ est identique, excepté qu'elle a un 2. Nous pouvons additionner ces rangées pour obtenir le chiffre cible 4, mais uniquement si le sous-ensemble $S'$ inclut au moins un littéral de $C_j$. De ces rangées de marge, celle qui doit être ajoutée dépend du nombre de littéraux de la clause $C_j$ inclus dans $S'$. Si $S'$ comprend un seul littéral, alors deux rangées sont nécessaires, car la somme dans la colonne est égale à 1 pour le littéral, plus 1 pour $q_j$ et plus 2 pour $q'_j$. Si $S'$ comprend deux littéraux, alors seul $q'_j$ est nécessaire, car la somme des colonnes est 2 pour les deux littéraux, plus 2 pour $q'_j$. Si $S'$ comprend trois littéraux, alors seul $q_j$ est nécessaire, car la somme de colonne est 3 pour les trois littéraux, plus 1 pour $q_j$. En revanche, si aucun littéral de la clause $C_j$ n'est inclus dans $S'$, alors $q_j + q'_j = 3$ ne suffit pas pour atteindre 4, le chiffre cible. Dans ce cas, nous pouvons y parvenir pour chaque clause uniquement si un littéral de la clause est inclus dans le sous-ensemble $S'$.

Nous avons vu la réduction et nous constatons qu'elle se fait en un temps polynomial. Nous créons $2n + 2k + 1$ entiers (la cible $t$ comprise), chacun de $n + k$ chiffres. En étudiant la table, vous pouvez voir que des entiers constitués, aucuns ne sont égaux et $S$ est réellement un ensemble (la définition d'un ensemble interdit tout doublon).

Copyright © 2013 Dunod.

Pour montrer que la réduction fonctionne, nous devons montrer que la formule 3-CNF $F$ a une affectation satisfaisante si et seulement s'il existe un sous-ensemble $S'$ de $S$ dont la somme des éléments est égale à $t$. À ce stade, vous avez compris l'idée, mais résumons-la. Tout d'abord, supposons qu'il existe une affectation satisfaisante pour $F$. Si cette affectation fixe $v_i$ à 1, alors nous incluons $x_i$ dans $S'$ ; sinon nous incluons $x'_i$. Puisque $x_i$ ou $x'_i$ se trouve dans $S$, la colonne de $v_i$ doit donner une somme égale à 1, correspondant au chiffre approprié de $t$. Puisque l'affectation satisfait chaque clause $C_j$, les rangées $x_i$ et $x'_i$ doivent apporter 1, 2 ou 3 (le nombre de littéraux de $C_j$ qui valent 1) à la somme dans la colonne de $C_j$. Nous ajoutons les rangées de marge $q_j$ et/ou $q'_j$ dans $S'$ afin d'atteindre le chiffre cible 4.

Réciproquement, supposons que la somme des éléments d'un sous-ensemble $S'$ de $S$ soit égale à $t$. Pour que $t$ ait un 1 dans les $n$ positions de gauche, $S'$ doit inclure uniquement $x_i$ ou $x'_i$ pour chaque variable $v_i$. S'il s'agit de $x_i$, alors nous fixons $v_i$ à 1. S'il s'agit de $x'_i$, alors nous fixons $v_i$ à 0. Puisque les rangées de marge $q_j$ et $q'_j$ additionnées ne permettent pas d'atteindre le chiffre cible 4 dans la colonne de la clause $C_j$, le sous-ensemble $S'$ doit également inclure au moins une rangée $x_i$ ou $x'_i$ avec un 1 dans la colonne de $C_j$. S'il inclut $x_i$, alors le littéral $v_i$ apparaît dans la clause $C_j$ et celle-ci est satisfaite. Si $S'$ inclut $x'_i$, alors le littéral NON $v_i$ apparaît dans la clause $C_j$, qui est satisfaite. Par conséquent, chaque clause est satisfaite et il existe une affectation satisfaisante pour la formule 3-CNF $F$.

Nous voyons donc que si nous pouvons résoudre le problème de somme de sous-ensemble en un temps polynomial, nous pouvons également déterminer en un temps polynomial si une formule 3-CNF peut être satisfaite. Puisque la satisfaisabilité 3-CNF est un problème NP-complet, c'est également le cas du problème de somme de sous-ensemble. De plus, si nous connaissons les entiers de l'ensemble $S$ dont la somme est égale à la cible $t$, alors nous pouvons déterminer les valeurs des variables de la formule 3-CNF afin que son évaluation donne 1.

Pour finir, signalons que, dans la réduction utilisée, les chiffres ne sont pas nécessairement des décimaux. L'important est qu'il n'y ait pas de retenue lors de l'addition des entiers. Puisqu'aucune somme des colonnes ne peut dépasser 6, il est possible d'interpréter les nombres dans n'importe quelle base supérieure ou égale à 7. L'exemple donné à la page 205 est tiré des valeurs de la table, mais interprétées en base 7.

## Partitionnement

Le **problème de partitionnement** est étroitement lié au problème de la somme de sous-ensemble. En réalité, il s'agit d'un cas particulier de ce problème : si $z$ est égal à la somme de tous les entiers de l'ensemble $S$, alors la cible $t$ est $z / 2$. Autrement dit, l'objectif est de déterminer s'il existe un partitionnement de l'ensemble $S$ en deux ensembles $S'$ et $S''$ tels que

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

chaque entier de $S$ soit dans $S'$ ou dans $S''$ mais pas dans les deux (voilà pourquoi $S'$ et $S''$ partitionnent S) et la somme des entiers dans $S'$ est égale à celle des entiers dans $S''$. Comme dans le problème de la somme de sous-ensemble, un certificat est un sous-ensemble de $S$.

Pour montrer que le problème de partitionnement est NP-difficile, nous effectuons, sans surprise, une réduction à partir du problème de la somme de sous-ensemble. Étant donné un ensemble $R$ d'entiers strictement positifs et un entier cible strictement positif $t$ en entrée du problème de la somme de sous-ensemble, nous construisons en un temps polynomial un ensemble $S$ pour l'entrée du problème de partitionnement. Nous commençons par calculer $z$ en tant que somme de tous les entiers de $R$. Nous supposons que $z$ est différent de $2t$, car, dans le cas contraire, le problème est déjà un problème de partitionnement. (Si $z = 2t$, alors $t = z / 2$ et nous tentons de trouver un sous-ensemble de $R$ dont la somme des éléments est égale à la somme des entiers qui ne sont pas dans le sous-ensemble.) Nous choisissons ensuite n'importe quel entier $y$ supérieur à $t + z$ et à $2z$. Nous définissons l'ensemble $S$ de sorte qu'il comprenne tous les entiers de $R$ et deux entiers supplémentaires, $y - t$ et $y - z + t$. Puisque $y$ est supérieur à $t + z$ et à $2z$, nous savons que $y - t$ et $y - z + t$ sont supérieurs à $z$ (la somme des entiers dans $R$) et ces deux entiers ne peuvent donc pas être dans $R$. (N'oubliez pas que puisque $S$ est un ensemble, tous ses éléments doivent être uniques. Nous savons également que, puisque $z$ est différent de $2t$, nous devons avoir $y - t \neq y - z + t$, et les deux nouveaux entiers sont donc uniques.) Notez que la somme de tous les entiers dans $S$ est égale à $z + (y - t) + (y - z + t)$, c'est-à-dire $2y$. Par conséquent, si $S$ est partitionné en deux sous-ensembles disjoints de somme égale, la somme de chaque sous-ensemble doit être égale à $y$.

Pour montrer que la réduction fonctionne, nous devons montrer qu'il existe un sous-ensemble $R'$ de $R$ dont la somme des entiers est égale à $t$ si et seulement s'il existe un partitionnement de $S$ en $S'$ et $S''$ tel que la somme des entiers dans $S'$ et celle des entiers dans $S''$ sont égales. Supposons tout d'abord qu'un sous-ensemble $R'$ de $R$ soit tel que la somme de ses entiers soit égale à $t$. Alors, les entiers de $R$ qui ne sont pas dans $R'$ doivent avoir une somme égale à $z - t$. Définissons l'ensemble $S'$ qui comprend tous les entiers de $R'$ ainsi que $y - t$ ($S''$ est donc constitué de $y - z + t$ et de tous les entiers de $R$ qui ne sont pas dans $R'$). Nous devons simplement montrer que les entiers dans $S'$ ont une somme égale à $y$. Rien n'est plus facile : la somme des entiers dans $R'$ est égale à $t$ et l'addition avec $y - t$ donne une somme égale à $y$.

Réciproquement, supposons qu'il existe un partitionnement de $S$ en $S'$ et $S''$, tous deux avec des éléments dont la somme est égale à $y$. Nous affirmons que les deux entiers ajoutés à $R$ lors de la construction de $S$ ($y - t$ et $y - z + t$) ne peuvent pas se trouver tous les deux dans $S'$ ni dans $S''$. En

Copyright © 2013 Dunod.

effet, s'ils étaient dans le même ensemble, la somme des éléments de cet ensemble serait au moins égale à $(y - t) + (y - z + t)$, c'est-à-dire $2y - z$. Mais rappelons que $y$ est supérieur à $z$ (et même à 2z). Par conséquent, $2y - z$ est supérieur à $y$. Donc, si $y - t$ et $y - z + t$ étaient dans le même ensemble, la somme des éléments de celui-ci serait supérieure à $y$. Nous savons donc que $y - t$ ou $y - z + t$ est dans $S'$ et que l'autre est dans $S''$. Puisque l'ensemble dans lequel se trouve $y - t$ n'a pas d'importance, supposons qu'il s'agisse de $S'$. Nous savons à présent que la somme des entiers dans $S'$ est égale à $y$, ce qui signifie que les entiers dans $S'$ autres que $y - t$ ont une somme égale à $y - (y - t)$, c'est-à-dire $t$. Puisque $y - z + t$ ne peut pas être également dans $S'$, nous savons que tous les autres entiers de $S'$ proviennent de $R$. Par conséquent, il existe un sous-ensemble de $R$ dont la somme des éléments est égale à $t$.

## Sac à dos

Dans le **problème du sac à dos**, nous avons un ensemble de $n$ éléments, chacun avec un poids et une valeur. La question est de savoir s'il existe un sous-ensemble d'éléments dont le poids total est inférieur ou égal à un poids $W$ donné et dont la valeur totale est supérieure ou égale à une valeur $V$ donnée. Il s'agit de la version décision d'un problème d'optimisation dans lequel l'objectif est de charger un sac à dos avec le sous-ensemble d'éléments de plus grande valeur, sans dépasser un poids limite. Ce problème d'optimisation a des applications évidentes, comme le choix des articles à emporter en randonnée ou, pour un cambrioleur, des biens à dérober.

Le problème de partitionnement n'est qu'un cas particulier du problème du sac à dos, dans lequel la valeur de chaque élément est égale à son poids et $W$ et $V$ sont égaux à la moitié du poids total. Si nous pouvons résoudre le problème du sac à dos en un temps polynomial, alors nous pouvons résoudre le problème de partitionnement en un temps polynomial. Par conséquent, le problème du sac à dos est au moins aussi difficile que le problème de partitionnement et il est donc inutile de passer par le processus complet de réduction pour montrer qu'il est NP-complet.

# Stratégies générales

Vous l'avez à présent probablement compris, il n'existe aucune solution générale pour réduire un problème en un autre de façon à prouver qu'il est NP-difficile. Certaines réductions sont relativement simples, par exemple celle du problème de circuit hamiltonien en problème du voyageur de commerce, alors que d'autres sont extrêmement compliquées. Voici quelques points à ne pas oublier et quelques stratégies qui peuvent souvent aider.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2013 Dunod.

## Du général au spécifique

Lors de la réduction d'un problème $X$ en un problème $Y$, vous devez toujours démarrer avec une entrée quelconque pour le problème $X$. En revanche, vous pouvez limiter l'entrée du problème $Y$ autant que vous voulez. Par exemple, lors de la réduction du problème de satisfaisabilité 3-CNF en problème de somme de sous-ensemble, la réduction devait traiter en entrée n'importe quelle formule 3-CNF, mais l'entrée produite pour la somme de sous-ensemble avait une structure spécifique : un ensemble de $2n + 2k$ entiers, chacun étant constitué d'une manière particulière. La réduction n'était pas en mesure de produire toutes les entrées possibles pour le problème de somme de sous-ensemble, mais cela n'avait pas d'importance. L'essentiel était que nous pouvions résoudre un problème de satisfaisabilité 3-CNF en transformant son entrée en une entrée pour le problème de somme de sous-ensemble et en utilisant ensuite la réponse à ce problème comme réponse au problème de satisfaisabilité 3-CNF.

Sachez cependant que chaque réduction doit être de la forme suivante : transformer n'importe quelle entrée du problème $X$ en entrée du problème $Y$, même avec un chaînage des réductions. Si vous voulez réduire un problème $X$ en un problème $Y$ et ce problème $Y$ en un problème $Z$, la première réduction doit transformer n'importe quelle entrée de $X$ en une entrée de $Y$, et la seconde doit transformer n'importe quelle entrée de $Y$ en une entrée de $Z$. Il ne suffit pas que la seconde réduction transforme uniquement les types d'entrées de $Y$ produits par la réduction à partir de $X$.

## Exploiter les restrictions dans le problème source de la réduction

En général, lors d'une réduction du problème $X$ en un problème $Y$, vous pouvez choisir le problème $X$ de façon à imposer des restrictions supplémentaires sur son entrée. Par exemple, il est presque toujours plus facile d'effectuer une réduction à partir d'une satisfaisabilité 3-CNF qu'à partir du problème mère de satisfaisabilité d'une formule booléenne. Les formules booléennes peuvent être arbitrairement compliquées, alors que nous avons vu comment exploiter la structure d'une formule 3-CNF pour la réduction.

De manière comparable, il est généralement plus évident d'effectuer une réduction à partir d'un problème de circuit hamiltonien qu'à partir d'un problème de voyageur de commerce, même s'ils semblent similaires. En effet, dans le problème du voyageur de commerce, le poids des arêtes peut être tout entier strictement positif, pas uniquement les 0 ou 1 requis lors de la réduction. Le problème de circuit hamiltonien est plus limité car chaque arête ne peut avoir que deux « valeurs » : présente ou absente.

Copyright © 2013 Dunod.

## Rechercher des cas particuliers

De nombreux problèmes NP-complets ne sont que des cas particuliers d'autres problèmes NP-complets, à l'instar du problème de partitionnement qui est un cas particulier du problème du sac à dos. Si vous savez que le problème $X$ est NP-complet et qu'il s'agit d'un cas particulier du problème $Y$, alors le problème $Y$ doit également être NP-complet. En effet, comme nous l'avons vu pour le problème du sac à dos, une solution à temps polynomial pour le problème $Y$ donne automatiquement une solution à temps polynomial pour le problème $X$. De façon plus intuitive, un problème $Y$ étant plus général qu'un problème $X$, il est au moins aussi difficile.

## Choisir un problème approprié pour la réduction

Une bonne stratégie consiste généralement à effectuer une réduction à partir d'un problème du même domaine, tout au moins d'un domaine lié, que le problème dont vous souhaitez prouver la NP-complétude. Par exemple, nous avons montré que le problème de couverture de sommets – un problème de graphe – était NP-complet en effectuant une réduction à partir du problème de la clique – également un problème de graphe. L'arbre généalogique de NP-complétude a montré que nous pouvons ensuite effectuer une réduction vers le problème de circuit hamiltonien, puis celui de chemin hamiltonien, puis celui du voyageur de commerce, puis celui de plus long chemin acyclique, tous des problèmes de graphe.

Cependant, il sera parfois préférable de passer d'un domaine à un autre, comme lorsque nous avons procédé à une réduction à partir du problème de satisfaisabilité 3-CNF en problème de la clique ou de la somme de sous-ensemble. La satisfaisabilité 3-CNF se révèle souvent un bon choix pour les réductions entre domaines.

Dans les problèmes de graphe, si vous devez sélectionner une partie du graphe, sans tenir compte de l'ordre, alors le problème de couverture de sommets est souvent un bon point de départ. Si l'ordre est important, tournez-vous vers le problème de circuit hamiltonien ou de chemin hamiltonien.

## Grandes récompenses et grandes pénalités

Lorsque nous avons transformé le graphe d'entrée $G$ pour le problème de circuit hamiltonien en graphe $G'$ pour le problème du voyageur de commerce, nous avons voulu privilégier l'utilisation des arêtes présentes dans $G$ lors du choix des arêtes de la tournée du voyageur de commerce. Pour cela, nous avons donné à ces arêtes un poids très faible : 0. Autrement dit, nous avons récompensé l'utilisation de ces arêtes.

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Une autre solution aurait été de donner aux arêtes de $G$ un poids fini et aux arêtes absentes de $G$ un poids infini. Dans ce cas, nous pénalisions l'utilisation des arêtes qui ne se trouvaient pas dans $G$. Si nous avions pris cette approche et avions donné à chaque arête de $G$ un poids $W$, alors nous aurions fixé le poids cible $k$ de la tournée complète du voyageur de commerce à $nW$.

### Concevoir des gadgets

Nous n'avons pas décrit la conception des gadgets, car ils peuvent être complexes. Les gadgets se révéleront utiles pour garantir certaines propriétés. Dans les livres cités dans la section « Lectures complémentaires », vous trouverez des exemples de construction et d'utilisation des gadgets dans les réductions.

## Perspectives

Vous trouvez probablement que l'image que nous avons dépeinte est plutôt sombre. Imaginons que vous essayiez d'arriver à un algorithme à temps polynomial pour résoudre un problème et que, quel que soit le bout par lequel vous le prenez, vous ne réussissez pas. Après un certain temps, vous êtes content d'arriver à un algorithme avec un temps $O(n^5)$, même si vous savez pertinemment que la croissance de $n^5$ est plutôt rapide. Ce problème est peut-être proche d'un autre que vous savez facilement résolu en un temps polynomial (comme la satisfaisabilité 2-CNF par rapport à la satisfaisabilité 3-CNF, ou la tournée eulérienne par rapport au circuit hamiltonien), et vous trouvez qu'il est extrêmement frustrant de ne pas pouvoir adapter l'algorithme polynomial à votre problème. Vous finissez par suspecter que vous vous arrachez peut-être les cheveux à vouloir résoudre un problème NP-complet. Et voilà que vous êtes en mesure de réduire un problème NP-complet connu en votre problème et savez donc qu'il est NP-difficile.

Est-ce la fin de l'histoire ? N'y a-t-il aucun espoir que vous puissiez résoudre le problème en un temps raisonnable ?

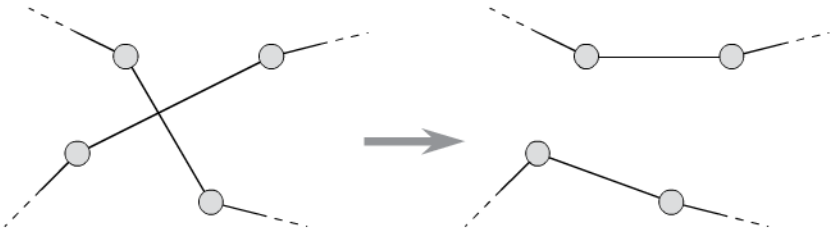
Pas vraiment. Lorsqu'un problème est NP-complet, cela signifie que certaines entrées posent des difficultés, mais pas nécessairement toutes. Par exemple, rechercher un plus long chemin acyclique dans un graphe orienté est NP-complet, mais si nous savons que le graphe est acyclique, alors nous pouvons trouver un plus long chemin acyclique non pas en un temps polynomial, mais en un temps $O(n + m)$ (le graphe comprend $n$ sommets et $m$ arêtes). Nous l'avons fait au Chapitre 5 pour la recherche d'un chemin critique dans un diagramme PERT. Prenons un autre exemple. Si vous essayez de résoudre le problème de partitionnement et si les entiers de

Copyright © 2013 Dunod.

l'ensemble ont une somme impaire, alors vous savez qu'il n'existe aucune solution pour obtenir deux partitions avec des sommes égales.

Les bonnes nouvelles vont au-delà de ces cas particuliers. À partir de maintenant, focalisons-nous sur des problèmes d'optimisation dont les versions décision sont NP-complètes, comme le problème du voyageur de commerce. Certaines méthodes rapides donnent de bons, et souvent de très bons, résultats. La technique de **séparation et évaluation** organise la recherche d'une solution optimale dans une arborescence. Elle coupe de grosses parties de l'arbre, réduisant ainsi l'espace de recherche, en se fondant sur l'idée simple que si elle peut déterminer que toutes les solutions qui émanent d'un nœud de l'arbre de recherche ne peuvent pas être meilleures que la meilleure solution trouvée jusqu'à présent, alors il est inutile de rechercher des solutions dans l'espace représenté par ce nœud et ses descendants.

Une autre technique souvent utile est la **recherche dans le voisinage**. Il s'agit de prendre une solution et d'appliquer des opérations locales afin d'essayer d'améliorer la solution jusqu'à ce que plus aucune amélioration ne se produise. Prenons le problème du voyageur de commerce dans lequel les sommets sont des points d'un plan et le poids de chaque arête correspond à la distance entre les points. Même avec ces restrictions, le problème reste NP-complet. Avec la technique **2-opt**, les arêtes qui se croisent sont basculées afin d'obtenir un circuit plus court :



Par ailleurs, des **algorithmes d'approximation** donnent des résultats qui sont garantis se trouver dans un certain facteur de la valeur optimale. Par exemple, si l'entrée du problème du voyageur de commerce respecte l'**inégalité triangulaire** – pour tout sommet $u$, $v$ et $x$, le poids de l'arête $(u,v)$ est inférieur ou égal à la somme des poids des arêtes $(u,x)$ et $(x,v)$ –, alors il existe un algorithme d'approximation simple qui permet de trouver une tournée dont le poids total est au maximum deux fois supérieur au poids le plus faible et cet algorithme s'exécute en un temps linéaire fonction de la taille de l'entrée. Il existe même un meilleur algorithme d'approximation polynomial pour ce cas, qui donne une tournée dont le poids total est inférieur ou égal à 3/2 fois le plus faible.

Assez étrangement, si deux problèmes NP-complets sont proches, un bon algorithme d'approximation peut produire une solution bonne pour l'un mais médiocre pour l'autre. Autrement dit, une solution quasi optimale

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

pour l'un des problèmes ne conduit pas nécessairement à une solution proche d'être quasi optimale pour l'autre.

Néanmoins, dans de nombreuses situations réelles, une solution presque optimale suffit. Si nous revenons à la société de livraison de colis aux camions marron, elle peut se satisfaire d'itinéraires quasi optimaux, même s'ils ne sont pas nécessairement les meilleurs possibles. Chaque euro économisé en planifiant des itinéraires plus efficaces améliorera ses bénéfices.

## Problèmes indécidables

Si vous avez l'impression que les problèmes NP-complets sont les plus difficiles dans le monde des algorithmes, vous risquez d'être surpris. Les théoriciens ont défini une vaste hiérarchie de classes de complexité qui dépendent du temps et des autres ressources nécessaires à résoudre un problème. Certains problèmes demandent un temps d'exécution qui varie de façon exponentielle avec la taille de l'entrée.

Mais la situation peut être pire encore. Dans le cas de certains problèmes, aucun algorithme n'est envisageable. Autrement dit, il existe des problèmes pour lesquels il a été prouvé qu'il est impossible de créer un algorithme qui donne une réponse exacte. Ces problèmes sont dits **indécidables**, le plus connu d'entre eux étant le **problème de l'arrêt**, que Alan Turing a prouvé indécidable en 1937. Dans ce problème, l'entrée est un programme informatique A avec une entrée $x$. L'objectif est de déterminer si un programme A, qui s'exécute sur l'entrée $x$, s'arrêtera un jour. Autrement dit, est-ce que A achèvera son traitement de l'entrée $x$ ?

Vous imaginez peut-être que vous pouvez écrire un programme, disons B, qui charge le programme A, lit $x$ et simule l'exécution de A sur $x$. Cette approche convient si A avec l'entrée de $x$ s'exécute jusqu'à achèvement. Que se passe-t-il dans le cas contraire ? Comment le programme B peut-il savoir qu'il peut déclarer que A ne s'arrêtera jamais ? B ne peut-il pas vérifier si A ne serait pas entré dans une sorte de boucle infinie ? Si vous pouvez effectivement écrire B de façon à vérifier certains cas dans lesquels A ne s'arrêtera pas, il est prouvé qu'il est impossible d'écrire un programme B qui s'arrête et qui indique correctement si A avec l'entrée $x$ s'arrêtera.

Puisqu'il n'est pas possible d'écrire un programme qui détermine si un autre programme qui s'exécute sur une entrée précise s'arrêtera un jour, il est également impossible d'écrire un programme qui détermine si un autre programme satisfait à ses spécifications. Comment un programme peut-il dire si un autre programme donne la bonne réponse s'il ne peut même pas dire si ce programme va s'arrêter ?

Copyright © 2013 Dunod.

Au cas où vous penseriez que les seuls problèmes indécidables ont trait aux propriétés des programmes informatiques, sachez que le **problème de correspondance de Post** (PCP, *Post's Correspondence Problem*) concerne les chaînes de caractères, comme celles rencontrées au Chapitre 7. Supposons que nous ayons au moins deux caractères et deux listes de $n$ chaînes, $A$ et $B$, constituées de ces caractères. Soit $A$ la liste des chaînes $A_1, A_2, A_3, \ldots, A_n$ et $B$ la liste des chaînes $B_1, B_2, B_3, \ldots, B_n$. Le problème est de déterminer s'il existe une séquence d'indices $i_1, i_2, i_3, \ldots, i_m$ tels que $A_{i1}A_{i2}A_{i3}\ldots A_{im}$ (c'est-à-dire la concaténation des chaînes $A_{i1}, A_{i2}, A_{i3}, \ldots, A_{im}$) donne la même chaîne que $B_{i1}Bi_2B_{i3}\ldots B_{im}$. Par exemple, supposons que les caractères soient `e`, `h`, `m`, `n`, `o`, `r` et `y`, que $n = 5$ et que :

$A_1 =$ `ey` $\quad B_1 =$ `ym`
$A_2 =$ `er` $\quad B_2 =$ `r`
$A_3 =$ `mo` $\quad B_3 =$ `oon`
$A_4 =$ `on` $\quad B_4 =$ `e`
$A_5 =$ `h` $\quad B_5 =$ `hon`

Alors, une solution est la séquence d'indices <5,4,1,3,4,2>, car $A_5A_4A_1A_3A_4A_2$ et $B_5B_4B_1B_3B_4B_2$ forment `honeymooner`. Bien entendu, s'il existe une solution, il en existe une infinité, car il suffit de répéter la séquence d'indices d'une solution (ce qui donne `honeymoonerhoneymooner`, etc.). Pour que le problème de correspondance de Post soit indécidable, nous devons autoriser l'utilisation à plusieurs reprises des chaînes de $A$ et de $B$, puisqu'il suffit sinon de lister toutes les combinaisons possibles des chaînes.

Bien que ce problème puisse ne pas sembler très intéressant en soi, nous pouvons le réduire en d'autres problèmes afin de montrer qu'ils sont eux aussi indécidables. L'idée est comparable à celle utilisée pour montrer qu'un problème est NP-difficile : étant donné une instance de PCP, la transformer en une instance d'un autre problème Q, de sorte que la réponse à l'instance de Q donne la réponse à l'instance de PCP. Si nous pouvons indiquer que Q est décidable, alors nous pouvons dire que PCP est décidable ; mais puisque nous savons que nous ne pouvons pas indiquer que PCP est décidable, alors Q doit être indécidable.

Parmi les problèmes indécidables dans lesquels nous pouvons réduire PCP, plusieurs d'entre eux concernent les **grammaires non contextuelles** (CFG, *Context-Free Grammar*) utilisées pour décrire la syntaxe de la plupart des langages de programmation. Une CFG est un ensemble de règles qui régissent la génération d'un **langage formel**, autrement dit un « ensemble de chaînes de caractères ». En effectuant une réduction vers PCP, nous pouvons prouver qu'il est impossible de décider si deux CFG produisent le même langage formel, si deux CFG génèrent des chaînes communes ou si une CFG est **ambiguë** (existe-t-il deux

Copyright © 2013 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

manières différentes de générer la même chaîne en utilisant les règles de la CFG ?).

## Conclusion

Nous venons de voir toute une variété d'algorithmes dans différents domaines. Nous avons étudié un algorithme qui prend un temps sous-linéaire : la recherche dichotomique. Nous avons examiné des algorithmes qui s'exécutent en un temps linéaire : recherche linéaire, tri par comptage, tri par base, tri topologique et recherche des plus courts chemins dans un graphe orienté acyclique. Nous avons rencontré des algorithmes qui ont un temps d'exécution $O(n \lg n)$ : tri par fusion et tri rapide (dans le cas moyen). Nous avons vu des algorithmes qui prennent un temps $O(n^2)$ : tri par sélection, tri par insertion et tri rapide (dans le cas le plus défavorable). Nous avons présenté des algorithmes sur les graphes qui ont des temps décrits par une combinaison non linéaire du nombre de sommets et d'arêtes : algorithme de Dijkstra et algorithme de Bellman-Ford. Nous avons détaillé un algorithme qui affiche un temps $\Theta(n^3)$ : l'algorithme de Floyd-Warshall. Nous avons découvert que, pour certains problèmes, nous ne savons pas si un algorithme à temps polynomial peut exister. Et nous venons même d'apprendre que, pour certains problèmes, aucun algorithme n'est possible, quel que soit son temps d'exécution.

Cette introduction au monde des algorithmes informatiques a été relativement brève[1], mais nous avons néanmoins couvert une zone étendue. Pourtant, cet ouvrage ne s'est intéressé qu'à une fine tranche du sujet. Par ailleurs, les analyses se sont limitées à un modèle informatique particulier, dans lequel un seul processeur effectue les opérations et où le temps de chaque opération est plus ou moins identique, que les données résident ou non en mémoire. Au fil des ans, de nombreux autres modèles informatiques ont été proposés, comme ceux fondés sur des multiprocesseurs, sur un temps d'exécution d'une opération variable selon l'emplacement des données, sur des données qui arrivent sous forme d'un flux non reproductible et sur un ordinateur quantique.

Par ailleurs, vous avez pu constater que ce domaine des algorithmes informatiques laisse de nombreuses questions sans réponse, ainsi que des questions qui n'ont pas encore été posées. Suivez des cours d'algorithmique, par exemple en ligne, et n'hésitez pas à nous aider !

1. Si nous comparons la taille de cet ouvrage avec celle du CLRS, dont la troisième édition fait environ 1 200 pages.

Copyright © 2013 Dunod.

Tri-Rapide, procédure 52
Tri-Sélection, procédure 35
Tri-Topologique, procédure 80
Tri-Très-Simple, procédure 64
type de donnée abstrait (ADT) 102

## U

Unicode 146

## V

valeur
  affecter à une variable 14
  de retour d'une procédure 12
variable
  dans une procédure 14
  de boucle 14
vérification en temps polynomial 189

© Dunod – Toute reproduction non autorisée est un délit.